Lecture Notes in Computer Science 6307

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Somesh Jha Robin Sommer
Christian Kreibich (Eds.)

Recent Advances in Intrusion Detection

13th International Symposium, RAID 2010
Ottawa, Ontario, Canada, September 15-17, 2010
Proceedings

Springer

Volume Editors

Somesh Jha
University of Wisconsin
Computer Sciences Department
Madison, WI 53706, USA
E-mail: jha@cs.wisc.edu

Robin Sommer
Christian Kreibich
International Computer Science Institute
1947 Center Street, Suite 600, Berkeley, CA 94704, USA
E-mail: {robin, christian}@icir.org

Library of Congress Control Number: 2010933245

CR Subject Classification (1998): C.2, K.6.5, D.4.6, E.3, H.4, I.2

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN 0302-9743
ISBN-10 3-642-15511-1 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-15511-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

On behalf of the Program Committee, it is our pleasure to present the proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection Systems (RAID 2010), which took place in Ottawa, Ontario, Canada, during September 15-17, 2010. As in the past, the symposium brought together leading researchers and practitioners from academia, government, and industry to discuss intrusion detection research and practice. There were eight technical sessions presenting full research papers on network protection, high performance, malware detection and defense (2 sessions), evaluation, forensics, anomaly detection and access protection, and Web security. Furthermore, there was a poster session presenting emerging research areas and case studies.

The RAID 2010 Program Committee received 102 full-paper submissions from all over the world. All submissions were carefully reviewed by independent reviewers on the basis of technical quality, topic, space, and overall balance. The final decision took place at a Program Committee meeting held during May 19-20 in Oakland, California, where 24 papers were eventually selected for presentation at the conference and publication in the proceedings. As a continued feature, the symposium later also accepted 15 poster presentations reporting early-stage research, demonstration of applications, or case studies. The authors of accepted posters were also offered the opportunity to have an extended abstract of their work included in the proceedings.

The success of RAID 2010 depended on the joint effort of many people. We would like to thank all the authors of submitted papers and posters. We would also like to thank the Program Committee members and additional reviewers, who volunteered their time to evaluate the numerous submissions. In addition, we would like to thank the General Chair, Frédéric Massicotte, for handling the conference arrangements; Christian Kreibich for handling the publication process; Thorsten Holz for publicizing the conference; Marc Grégoire for finding sponsors for the conference; and the Communications Research Centre Canada for maintaining the conference website. Finally, we would like to thank our sponsors for their support.

July 2010

Somesh Jha
Robin Sommer

Organization

Organizing Committee

General Chair	Frédéric Massicotte, Communications Research Centre Canada
General Co-chair	Marc Grégoire, Defence Research and Development Canada
Program Chair	Somesh Jha, University of Wisconsin, USA
Program Co-chair	Robin Sommer, ICSI / LBNL, USA
Sponsorship Chair	Marc Grégoire, Defence Research and Development Canada
Publication Chair	Christian Kreibich, ICSI, USA
Publicity Chair	Thorsten Holz, Technical University Vienna, Austria

Program Committee

Michael Bailey	University of Michigan, USA
Davide Balzarotti	Eurecom, France
Adam Barth	UC Berkeley, USA
David Brumley	Carnegie Mellon University, USA
Mihai Christodorescu	IBM T.J. Watson Research Center, USA
Manuel Costa	Microsoft Research, Cambridge, UK
Jonathan Giffin	Georgia Institute of Technology, USA
Guofei Gu	Texas A & M University, USA
Thorsten Holz	Technical University Vienna, Austria
Jaeyeon Jung	Intel Research, USA
Christian Kreibich	International Computer Science Institute, USA
Wenke Lee	Georgia Institute of Technology, USA
Corrado Leita	Symantec Research Europe, France
Gregor Maier	TU Berlin / Deutsche Telekom Laboratories, Germany
Benjamin Morin	Central Directorate for Information System Security, France
Phil Porras	SRI International, USA
Anil Somayaji	Carleton University, Canada
V.N. Venkatkrishnan	University of Illinois (Chicago), USA
Charles Wright	MIT Lincoln Laboratory, USA
Vinod Yegnewswaran	SRI International, USA

External Reviewers

Manos Antonakakis
Prithvi Bisht
Kevin Carter
Byung-gon Chun
Chris Connelly
Loic Duflot
Ashish Gehani
Kalpana Gondi
Christian Gorecki
Ralf Hund
Clemens Kolbitsch
Oleg Krogius
Andrea Lanzi
Xiapu Luo
Ludovic Mé
Paolo Milani Comparetti
Andreas Moser
Collin Mulliner
Kaustubh Nyalkalkar
Jon Oberheide
Roberto Perdisci
Fabien Pouget
Konrad Rieck
Hassen Saidi
Monirul Sharif
Seungwon Shin
Kapil Singh
Brad Spengler
Gianluca Stringhini
Mike Ter Louw
Yohann Thomas
Elvis Tombini
Carsten Willems
Yunjing Xu
Chao Yang
Michelle Zhou

Steering Committee

Chair Marc Dacier, Symantec Research Europe
Members Hervé Debar, France Telecom R&D, France
Deborah Frincke, Pacific Northwest National Lab, USA
Ming-Yuh Huang, The Boeing Company, USA
Erland Jonsson, Chalmers University of Technology, Sweden
Engin Kirda, Institute Eurecom, France
Wenke Lee, Georgia Institute of Technology, USA
Ludovic Mé, Supélec, France
Alfonso Valdes, SRI International, USA
Giovanni Vigna, UC Santa Barbara, USA
Andreas Wespi, IBM Research, Switzerland
S. Felix Wu, UC Davis, USA
Diego Zamboni, HP Professional Services, Mexico
Christopher Kruegel, UC Santa Barbara, USA

Table of Contents

Network Protection

What Is the Impact of P2P Traffic on Anomaly Detection? 1
Irfan Ul Haq, Sardar Ali, Hassan Khan, and Syed Ali Khayam

A Centralized Monitoring Infrastructure for Improving DNS Security ... 18
Manos Antonakakis, David Dagon, Xiapu Luo, Roberto Perdisci, Wenke Lee, and Justin Bellmor

Behavior-Based Worm Detectors Compared 38
Shad Stafford and Jun Li

High Performance

Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams 58
Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith

GrAVity: A Massively Parallel Antivirus Engine 79
Giorgos Vasiliadis and Sotiris Ioannidis

Malware Detection and Defence

Automatic Discovery of Parasitic Malware 97
Abhinav Srivastava and Jonathon Giffin

BotSwindler: Tamper Resistant Injection of Believable Decoys in VM-Based Hosts for Crimeware Detection 118
Brian M. Bowen, Pratap Prabhu, Vasileios P. Kemerlis, Stelios Sidiroglou, Angelos D. Keromytis, and Salvatore J. Stolfo

CANVuS: Context-Aware Network Vulnerability Scanning 138
Yunjing Xu, Michael Bailey, Eric Vander Weele, and Farnam Jahanian

HyperCheck: A Hardware-Assisted Integrity Monitor 158
Jiang Wang, Angelos Stavrou, and Anup Ghosh

Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory 178
Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang

Bait Your Hook: A Novel Detection Technique for Keyloggers 198
Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo

Evaluation

Generating Client Workloads and High-Fidelity Network Traffic for Controllable, Repeatable Experiments in Computer Security 218
Charles V. Wright, Christopher Connelly, Timothy Braje, Jesse C. Rabek, Lee M. Rossey, and Robert K. Cunningham

On Challenges in Evaluating Malware Clustering 238
Peng Li, Limin Liu, Debin Gao, and Michael K. Reiter

Why Did My Detector Do *That*?! Predicting Keystroke-Dynamics Error Rates 256
Kevin Killourhy and Roy Maxion

Forensics

NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring 277
Paul Giura and Nasir Memon

Live and Trustworthy Forensic Analysis of Commodity Production Systems 297
Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro

Hybrid Analysis and Control of Malware 317
Kevin A. Roundy and Barton P. Miller

Anomaly Detection

Anomaly Detection and Mitigation for Disaster Area Networks 339
Jordi Cucurull, Mikael Asplund, and Simin Nadjm-Tehrani

Community Epidemic Detection Using Time-Correlated Anomalies 360
Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken

A Data-Centric Approach to Insider Attack Detection in Database Systems 382
Sunu Mathew, Michalis Petropoulos, Hung Q. Ngo, and Shambhu Upadhyaya

Privilege States Based Access Control for Fine-Grained Intrusion Response 402
Ashish Kamra and Elisa Bertino

Web Security

Abusing Social Networks for Automated User Profiling 422
Marco Balduzzi, Christian Platzer, Thorsten Holz, Engin Kirda, Davide Balzarotti, and Christopher Kruegel

An Analysis of Rogue AV Campaigns 442
Marco Cova, Corrado Leita, Olivier Thonnard, Angelos D. Keromytis, and Marc Dacier

Fast-Flux Bot Detection in Real Time 464
Ching-Hsiang Hsu, Chun-Ying Huang, and Kuan-Ta Chen

Posters

A Client-Based and Server-Enhanced Defense Mechanism for Cross-Site Request Forgery 484
Luyi Xing, Yuqing Zhang, and Shenlong Chen

A Distributed Honeynet at KFUPM: A Case Study 486
Mohammed Sqalli, Raed AlShaikh, and Ezzat Ahmed

Aspect-Based Attack Detection in Large-Scale Networks 488
Martin Drašar, Jan Vykopal, Radek Krejčí, and Pavel Čeleda

Detecting Network Anomalies in Backbone Networks 490
Christian Callegari, Loris Gazzarrini, Stefano Giordano, Michele Pagano, and Teresa Pepe

Detecting the Onset of Infection for Secure Hosts 492
Kui Xu, Qiang Ma, and Danfeng (Daphne) Yao

Eliminating Human Specification in Static Analysis 494
Ying Kong, Yuqing Zhang, and Qixu Liu

Evaluation of the Common Dataset Used in Anti-Malware Engineering Workshop 2009 496
Hosoi Takurou and Kanta Matsuura

Inferring Protocol State Machine from Real-World Trace 498
Yipeng Wang, Zhibin Zhang, and Li Guo

MEDUSA: Mining Events to Detect Undesirable uSer Actions in SCADA 500
Dina Hadžiosmanović, Damiano Bolzoni, and Pieter Hartel

On Estimating Cyber Adversaries' Capabilities: A Bayesian Model Approach 502
Jianchun Jiang, Weifeng Chen, and Liping Ding

Security System for Encrypted Environments (S2E2) 505
Robert Koch and Gabi Dreo Rodosek

Towards Automatic Deduction and Event Reconstruction Using Forensic Lucid and Probabilities to Encode the IDS Evidence 508
Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi

Toward Specification-Based Intrusion Detection for Web Applications ... 510
Salman Niksefat, Mohammad Mahdi Ahaniha, Babak Sadeghiyan, and Mehdi Shajari

Toward Whole-System Dynamic Analysis for ARM-Based Mobile Devices 512
Ryan Whelan and David Kaeli

Using IRP for Malware Detection 514
FuYong Zhang, DeYu Qi, and JingLin Hu

Author Index 517

What Is the Impact of P2P Traffic on Anomaly Detection?*

Irfan Ul Haq, Sardar Ali, Hassan Khan, and Syed Ali Khayam

School of Electrical Engineering & Computer Science
National University of Sciences & Technology (NUST)
Islamabad 44000, Pakistan
{irfan.haq,sardar.ali,hassan.khan,ali.khayam}@seecs.nust.edu.pk

Abstract. Recent studies estimate that peer-to-peer (p2p) traffic comprises 40-70% of today's Internet traffic [1]. Surprisingly, the impact of p2p traffic on anomaly detection has not been investigated. In this paper, we collect and use a labeled dataset containing diverse network anomalies (portscans, TCP floods, UDP floods, at varying rates) and p2p traffic (encrypted and unencrypted with BitTorrent, Vuze, Flashget, μTorrent, Deluge, BitComet, Halite, eDonkey and Kademlia clients) to empirically quantify the impact of p2p traffic on anomaly detection. Four prominent anomaly detectors (TRW-CB [7], Rate Limiting [8], Maximum Entropy [10] and NETAD [11]) are evaluated on this dataset.

Our results reveal that: 1) p2p traffic results in up to 30% decrease in detection rate and up to 45% increase in false positive rate; 2) due to a partial overlap of traffic behaviors, p2p traffic inadvertently provides an effective evasion cover for high- and low-rate attacks; and 3) training an anomaly detector on p2p traffic, instead of improving accuracy, introduces a significant accuracy degradation for the anomaly detector. Based on these results, we argue that only p2p traffic filtering can provide a pragmatic, yet short-term, solution to this problem. We incorporate two prominent p2p traffic classifiers (OpenDPI [23] and Karagiannis' Payload Classifier(KPC) [24]) as pre-processors into the anomaly detectors and show that the existing non-proprietary p2p traffic classifiers do not have sufficient accuracies to mitigate the negative impacts of p2p traffic on anomaly detection.

Given the premise that p2p traffic is here to stay, our work demonstrates the need to rethink the classical anomaly detection design philosophy with a focus on performing anomaly detection in the presence of p2p traffic. We make our dataset publicly available for evaluation of future anomaly detectors that are designed to operate with p2p traffic.

1 Introduction

During March of 2009, a record number of 4,543 anomalies was recorded by an open-source TRW-CB based [7] anomaly detector deployed on our school's

* This work is supported by Pakistan National ICT R&D Fund.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 1–17, 2010.

network. The network administrators took it as a result of a zero-day attack and updated the antivirus and antispyware definitions on school hosts. However, TRW-CB continued reporting anomalies even after the update. An investigation of this strange behavior by correlating the TRW-CB logs and the network logs revealed that the culprit was p2p traffic which was being reported as anomalous.[1] This strange behavior of TRW-CB was communicated to us which intrigued us to investigate the impact of p2p traffic on anomaly detection.

Based on the results of our investigation, in this paper we empirically answer the following open question: *How much perturbations are introduced in anomaly detection metrics by p2p traffic*[2] *and how can these perturbations be mitigated?* A general answer to this question can be inferred intuitively because some features of p2p traffic are quite similar to those of malicious traffic and quite different from the bulk of benign TCP traffic [5]. Hence, the accuracy of an anomaly detector, which flags deviations from a model of normal behavior, is bound to degrade in the presence of p2p traffic. For example, the decentralized and distributed nature of the p2p architecture results in establishment of a large number of connections to random ports during boot-strap which shares similarities with portscan attacks; compare a torrent client "scanning" over 50 peers during boot-strapping to `MyDoom-A` with an average scan rate of 9 scans per minute. Similarly, high churn rate in p2p networks results in a large number of failed connections[3] which is another commonly-observed phenomenon during portscan attacks.

While a general sense can be determined intuitively, our empirical study gives deeper insights by breaking the above question into the following set of important sub-questions: 1) How much degradation does p2p traffic induce in anomaly detection accuracy (detection and false positive rates)? 2) Which anomaly detection metrics/principles are more sensitive to p2p traffic and why? 3) Does the aggressive nature of p2p traffic dominate some/all attack classes and high-/low-rate attacks? 4) Can an anomaly detector handle p2p traffic if it is trained on a dataset containing p2p traffic? 5) Can a pragmatic solution be designed to make an anomaly detector insensitive to the p2p traffic? 6) Can existing public p2p traffic classifiers mitigate the degradation in anomaly detection accuracy? 7) What are the open problems in designing anomaly detectors which operate effectively in today's Internet traffic?

To empirically answer the above questions, we collect a labeled dataset containing diverse network anomalies (portscans, TCP floods, UDP floods, at varying rates) and p2p traffic (encrypted and unencrypted with BitTorrent, Vuze, Flashget, μTorrent, Deluge, BitComet, Halite, eDonkey and KAD clients). Since it is not possible to evaluate all existing anomaly detectors, ROC-based

[1] This sudden spike was caused by recent relocation of students' dormitories inside our newly-built campus and the students' usage of p2p applications in their dormitories.

[2] While our evaluations focus on p2p file sharing traffic, p2p VOIP and p2p streaming video traffic also exhibit similar traffic behaviors.

[3] Failed connections is a feature which is employed to detect malicious hosts [6]-[9] as well as p2p file sharing hosts [20],[21].

accuracies of four prominent anomaly detectors (TRW-CB [7], Rate Limiting [8], Maximum Entropy [10] and NETAD [11]) are evaluated on this dataset.

Our results reveal that *all* the anomaly detectors experience an unacceptable (up to 30%) drop in detection rates and a significant (up to 45%) increase in false alarm rates when operating with p2p traffic. Henceforth in the paper, we refer to this accuracy degradation as the *torrent effect* on anomaly detection. We evaluate the torrent effect by evaluating the anomaly detectors on different attack rates and classes. We show that anomaly detectors deliver varying accuracies on different attack classes and this varying performance is a function of the design principle of a given anomaly detectors. Similarly, we show that p2p traffic inadvertently acts as a very effective evasion cover for low-rate attacks as detection of such attacks is seriously affected by p2p traffic.

Based on the significant and consistent accuracy degradations observed in our study, we argue that a p2p traffic classifier based pre-processor can offer the anomaly detectors a pragmatic, albeit short-term, relief from the torrent effect.[4] By incorporating OpenDPI [23] into the IDSs we see 12% improvement in detection accuracy with 4% reduction in false positive rate. Similarly, incorporating KPC [24] results in 18% improvement in detection accuracy and a 48% reduction in false positive rate. However, even with these improvements, existing non-proprietary p2p traffic classifiers do not have sufficient traffic classification accuracies to eliminate the torrent effect.

Recent trends indicate that the volume of p2p traffic is reducing as service providers are now deploying commercial p2p traffic classifiers to throttle p2p traffic in real-time [1]-[4]. Nevertheless, due to the ubiquity and popularity of p2p networks and software, even with reduced-volumes, p2p traffic is anticipated to continue comprising a significant percentage of the Internet's traffic in the coming years [34]. We therefore advocate a fundamental rethinking of the anomaly detection design philosophy with future anomaly detectors catering for p2p traffic in their inherent design. We make our dataset publicly available for evaluation of such future anomaly detectors.

2 Related Work and Background

While significant research has recently been focused towards evaluating and understanding trends in anomaly detection [16], to the best of our knowledge, the impact of p2p traffic on intrusion detection has not been explored. Therefore, in this section we only provide a brief overview of the anomaly detectors evaluated in this work; interested readers are referred to the original papers [7],[8],[10] and [11] for detailed descriptions of the anomaly detectors.

The Rate Limiting approach [8], detects anomalous connection behavior by putting new connections exceeding a certain threshold in a queue. An alarm is raised when the queue length exceeds a threshold. TRW-CB [7] limits the rate at which new connections are initiated by applying the sequential hypothesis

[4] Commercial IDSs are already incorporating p2p traffic classifiers (DPI engines) into their products [31]-[33].

testing and by using a credit increase/decrease algorithm to slow down hosts that are experiencing unsuccessful connections. The Maximum Entropy based detector [10] estimates the benign traffic's baseline distribution using Maximum Entropy method by dividing the traffic into 2,348 packet classes. These packet classes are defined based on destination ports and the transport protocols. Kullback-Leibler (K-L) divergence measure is then used to flag anomalies if divergence from the baseline distribution exceeds a threshold from the baseline distribution. NETAD [11] operates on rule-based filtered traffic in a modeled subset of common protocols. It computes a packet score depending on the time and frequency of each byte of packet, and rare/novel header values are assigned high scores. A threshold is applied on a packet's score to find anomalous packets. For performance evaluations, parameter tuning for these anomaly detectors is performed in the same fashion as in a recent evaluation study [35].

We chose these anomaly detectors to ensure diversity because these detectors have very different detection principles and features, and operate at different traffic granularities. On the one hand, we use Rate Limiting [8] which is a connection-based programmed system using a thresholding approach, while, on the other hand, we use a statistical programmed system, TRW-CB [7]. Similarly, we employ an information-theoretic self-learning system like Maximum Entropy [10] as opposed to NETAD [11] which is a packet-based rule-modeling system.

3 Dataset Description

For the present problem, we wanted to use real, labeled and public background and attack datasets to measure the accuracy. Furthermore, for comprehensive evaluation, we needed attacks of different types (DoS, portscan, etc.) and different rates for each attack type. Finally, we needed labeled p2p traffic from various clients and p2p protocols in our dataset. While some old attack datasets are available [17]-[19], they do not contain p2p traffic and do not contain attacks of different types. Therefore, we collect our own network traffic dataset and make it publicly available for repeatable performance evaluations.[5] The rest of this section describes our data collection methodology.

We collect dataset in our campus network. The research labs in our campus are located in research wing and traffic from each research lab is relayed through a 3COM4500G switch to research wing's Cisco 3750 router using fiber connections, as shown in Figure 1. The wing router is connected to the distribution router which handles traffic of the entire campus. The research wing router routes traffic for over 50 computers. Three computers in our research lab were used to generate attack traffic. P2P traffic was generated by hosting p2p file sharing applications on twelve computers in different labs. Due to privacy constraints, we were only allowed to collect traces at the research wing router. We now provide the details for normal, p2p and attack traffic in our dataset.

[5] The dataset collected for this work is available at `http://wisnet.seecs.nust.edu.pk/projects/ENS/DataSets.html`

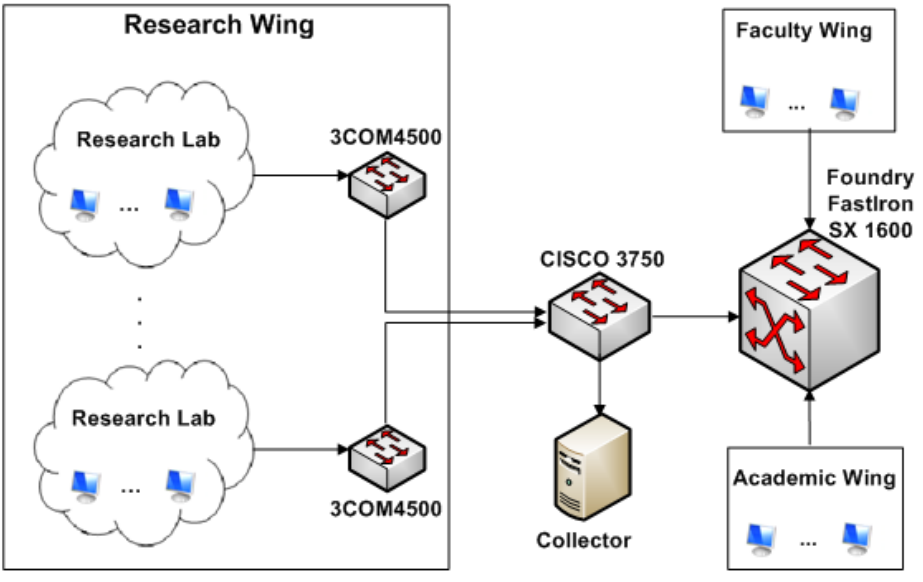


Fig. 1. Dataset collection setup

Table 1. P2P File Sharing Application Traffic Statistics

Client Name & Version	Sessions Estb.	Traffic Vol.	Throughput(Mbps)
Vuze 4.0	20	685 MB	0.8
Flashget 1.9.6	62	60.7 MB	1.2
UTorrent 1.8.1	30	1.08 GB	1.7
BitTorrent 6.1.2	134	1.59 GB	2.62
Deluge 1.0.7	30	171 MB	0.72
BitComet 1.07	20	57.4 MB	0.6
Halite 0.3.1	9	413 MB	0.94
eMule v0.49b	203	2.67 GB	1.2

3.1 Normal Traffic

We captured the normal traffic in six periods, each one of over three hours. During traffic capturing, different applications were hosted on the machines including file transfer, web browsing, instant messaging, real-time video streaming, etc. It was ensured that during normal traffic capturing, no p2p application was hosted on any of the client machines. The mean packet rate recorded for the background traffic was about 3168 pkts/sec and the standard deviation was 1683 pkts/sec.

3.2 P2P Traffic

The p2p traffic in our traces belongs to the BitTorrent, eDonkey and Kademlia protocols. These protocols were chosen as representative traffic from p2p traffic

Table 2. Attack Characteristics & Background Traffic Information During Attacks

Attack Name	Attack Characteristics	Attack Rate (pkts/sec)	Background Traffic Statistics at attack time (pkts/sec)	
			μ	σ
TCP-SYN portscans	Fixed src IP addr Two distinct attacks: First scan on port 80, Second scan on port 135	0.1	2462.9	474.4
		1	3002.6	398.0
		10	3325.2	397.7
		100	6100.0	2492.4
		1000	3084.7	247.4
TCP-SYN flood (DoS)	Two remote servers attacked Attacked ports: 143, 22, 138, 137, 21	0.1	2240.1	216.7
		1	2699.1	328.8
		10	4409.8	1666.2
		100	3964.1	1670.4
		1000	3000.9	238.0
UDP flood fraggle	Two remote servers attacked Attacked ports: 22, 80, 135, 143	0.1	2025.8	506.4
		1	2479.1	291.0
		10	4028.4	1893.1
		100	6565.7	3006.9
		1000	2883.7	260.8

class because these protocols generate the largest volumes of p2p traffic on Internet [1]. During our trace collection for BitTorrent protocol, we used multiple torrent files for transferring data from/to multiple geographical locations for each torrent session. Multiple clients including Vuze, Flashget, μTorrent, BitTorrent, Deluge, BitComet and Halite were used to introduce real-world diversity in the dataset as different clients might had different behavior. For eMule sessions two options related to protocol obfuscation and communication with obfuscated connections only ("Allow obfuscated connections only"), were enabled in the client to ensure logging of encrypted traffic. Similarly, encryption was enabled for the torrent sessions. Statistics for the p2p file sharing applications' traffic are given in Table 1.

3.3 Attack Traffic

For attack traffic, we launch port scans (TCP SYN), DoS (TCP SYN) and fraggle (UDP flood) simultaneously from three end hosts in our research lab. The DoS attacks was launched on two servers under our administration with public IP addresses. Each attack was launched for a period of five minutes with spoofed IP address. For every attack type, three low-rate ({0.1, 1, 10} pkts/sec) and two high-rate ({100, 1000} pkts/sec) instances were launched. The attack characteristics for each attack are shown in Table 2.

4 Investigating the Torrent Effect

We now embark on finding answers to the questions that were raised in the introduction. To this end, we evaluate the anomaly detectors on datasets with

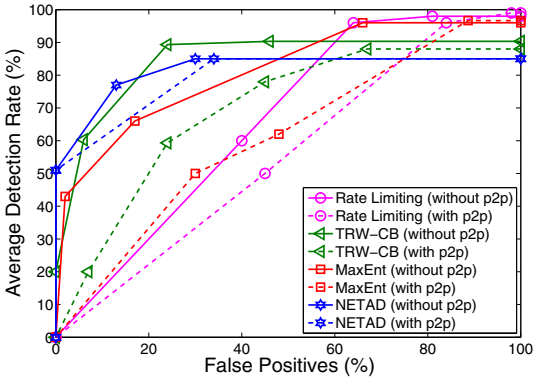


Fig. 2. ROC results to quantify the impact of p2p traffic on anomaly detection accuracy; each ROC point is averaged over 3(attacks) × 3(instances/attack) × 5(rates/instance) = 45 attack windows of 5 minutes each.

varying proportions of attack and p2p traffic. In this section, we perform the evaluations to find out the impact of p2p traffic on anomaly detection accuracy; its correlation with high- and low-rate attacks; its affect on different attack classes and whether p2p traffic should be used to train an anomaly detector. We defer the solution to the torrent effect to Section 5.

4.1 How Much Degradation Does p2p Traffic Induce in Anomaly Detection Accuracy?

We first investigate the impact of p2p traffic on the anomaly detectors' detection and false alarm rates. Figure 2 plots the Receiver Operating Characteristic (ROC) curves of the anomaly detectors on the dataset with p2p traffic and on the same dataset with p2p traffic removed from it. The anomaly detectors in this case were trained only on non-p2p traffic. With the introduction of p2p traffic, the detection rates of all anomaly detectors drop and the false positive rates increase. This behavior is observed because of the similarities between p2p and malicious traffic features, such as a large number of connection attempts, a large number of failed connections, and the use of unprivileged ports. Figure 2 shows that Maximum Entropy and TRW-CB fail miserably (up to 30% reduction in detection rate and up to 40% increase in false positives) when they operate on the dataset with p2p traffic. On the other hand, the detection rates of Rate Limiting and NETAD never degrade by more than 20% and 10%, respectively. Similarly, for Rate Limiting and NETAD, the average false positive rate increase remains around 10%. Deferring further discussion on relative degradation for each anomaly detector to the next section, we deduce from Figure 2 that the accuracies of *all* anomaly detectors degrade considerably due to p2p traffic.

4.2 Which Anomaly Detection Metrics/Principles Are More Sensitive to p2p Traffic and Why?

As we discussed in Section 2, we chose a diverse set of anomaly detectors which employ varying traffic features and operate on assorted detection principles. We now analyze the sensitivity of each detector to p2p traffic with a motivation to identify design guidelines to make these detectors insensitive to background traffic.

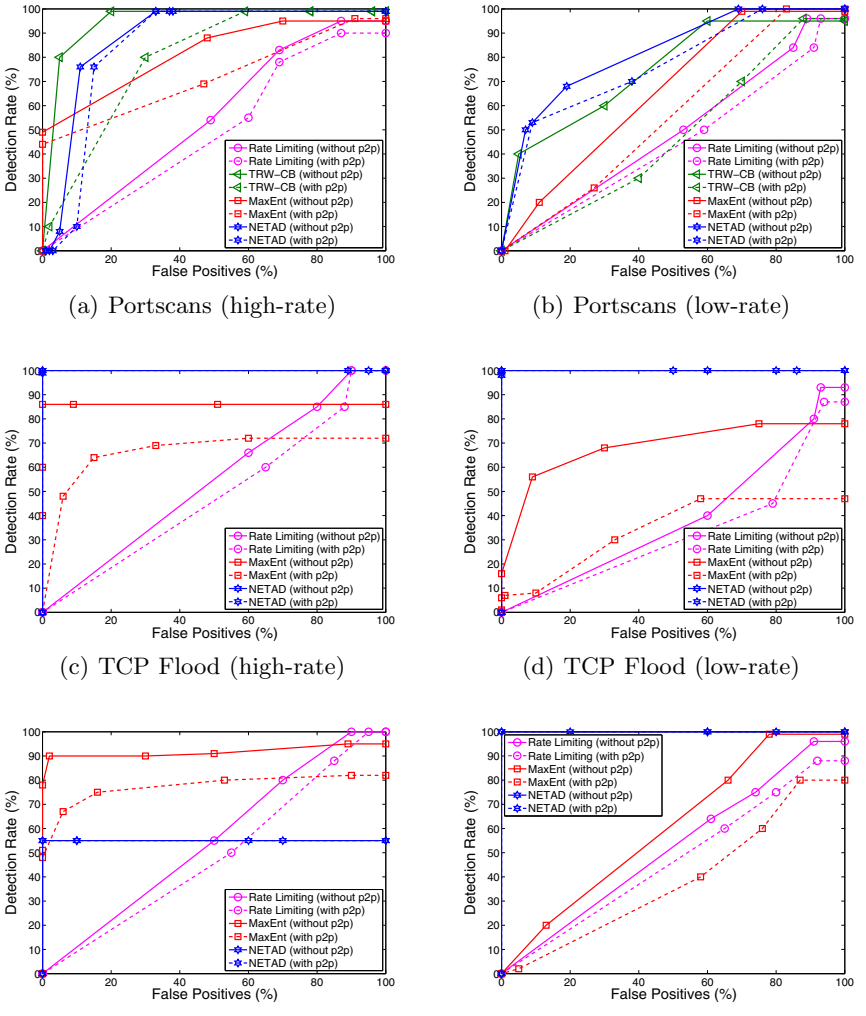
Figure 2 shows that NETAD provides the best overall accuracy and sustains it under p2p traffic. This is surprising because NETAD is in essence a rule-based detector and previous studies have shown that such algorithms fail in many attack scenarios [35,36]. Further investigation revealed that the graceful accuracy degradation of NETAD is mainly because of two rules that it uses to classify normal traffic: 1) All UDP traffic on higher ports (> 1023); 2) TCP data starting after 100 bytes. Both of these rules are satisfied by most of the p2p clients because the communication with trackers and peers takes place on higher ports, and TCP connections with each peer requires a sequence of TCP control packet exchanges to establish the number and sizes of file chunks to be downloaded. Due to these rules, NETAD continued to detect most of the p2p traffic as non-malicious.

While both Rate Limiting and TRW-CB use outgoing connections as the key detection feature, Figure 2 shows that Rate Limiting is less sensitive to p2p traffic as compared to TRW-CB. We noticed that the low sensitivity of Rate Limiting is because it operates on a long-term profile of traffic by keeping new connections in a queue. P2P applications establish a large number of connections, but in a short span of time during bootstrap. Therefore, Rate Limiting's queue threshold was not exceeded during this short-term connection activity period. On the other hand, the affect of p2p bootstrapping becomes very pronounced for TRW-CB which keeps changing its score with each individual connection attempt. Despite the low degradation observed in Rate Limiting, we note that the Rate Limiting detector generally provides the worst accuracy among all the evaluated detectors. Therefore, while its relative accuracy degradation in the presence of p2p traffic is low, its overall accuracy is considerably lower than TRW-CB; at 20% false positive rate, TRW-CB gives approximately 26% better detection rate than Rate Limiting. Hence, TRW-CB, despite having a larger accuracy degradation, should still be the preferred choice of portscan anomaly detector.

The accuracy degradation observed for Maximum Entropy is due to its reliance on a baseline distribution of destination port numbers. P2P peers generally use random port numbers which result in a distribution approaching uniformity which is incorrectly classified as malicious by the Maximum Entropy detector.

4.3 Does the Aggressive Nature of p2p Traffic Dominate Some/All Attack Classes and High-/Low-Rate Attacks?

We now move to the question about whether or not p2p traffic has the same impact on different attack classes and rates. To address this question, Figure 3



(a) Portscans (high-rate)

(b) Portscans (low-rate)

(c) TCP Flood (high-rate)

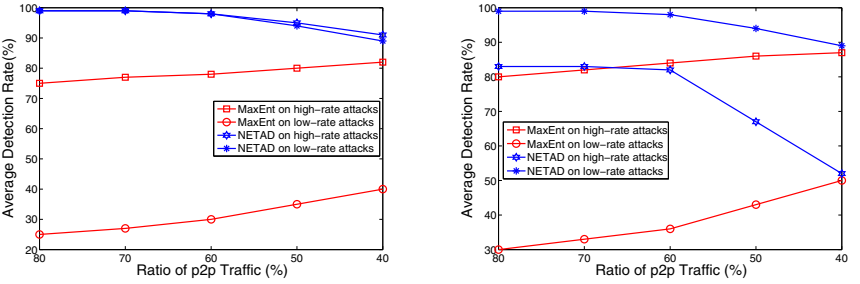
(d) TCP Flood (low-rate)

(e) UDP Flood (high-rate)

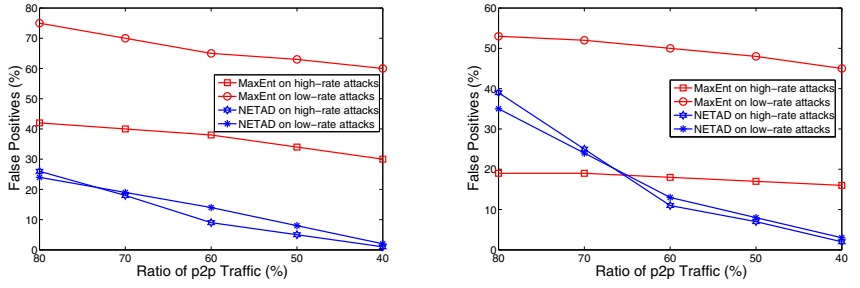
(f) UDP Flood (low-rate)

Fig. 3. ROCs for different attack classes/rates; results of TRW-CB for flooding attacks are omitted as it had 0% detection rate due to random source IP address spoofing used by flooding attacks.

plots separate ROCs for each attack class and rate. We note that the performance of NETAD does not degrade for flooding attacks when p2p traffic is introduced, but its accuracy degrades for portscans. On the other hand, performance penalty for Maximum Entropy in case of flooding attacks is much more than that for portscans. This is mainly because of the differing design principles of these anomaly detectors. Flooding attacks are detected by NETAD because the floods were launched on lower ports [Table 2], whereas p2p communication



(a) Dataset with p2p traffic (b) Dataset without p2p traffic



(c) Dataset with p2p traffic (d) Dataset without p2p traffic

Fig. 4. Results of training IDSs on p2p traffic

was using higher ports for communication. The same attacks degrade Maximum Entropy detector's accuracy because p2p traffic on higher ports increases the variance and entropy of the port distribution, thereby resulting in a large number of false positives from windows containing p2p activity. These results indicate that depending on the detection principles and features employed by an anomaly detector, the affect of p2p traffic can be much more pronounced for some attack classes.

We are also interested in determining how p2p traffic affects low- and high-rate attacks. From Figure 3, we observe that detection of low-rate attacks is much more seriously affected than that of high-rate attacks. Thus p2p traffic inadvertently acts as a very effective evasion cover for low-rate attacks. This evasion cover is also provided for high-rate attacks, but the cover is blown for the scenarios where the sustained attack connection rate overwhelms the short-term p2p connection burst.

4.4 Can an Anomaly Detector Handle p2p Traffic if It Is Trained on a Dataset Containing p2p Traffic?

Our performance evaluations thus far have indicated that the p2p traffic adversely affects the accuracies of *all* anomaly detectors evaluated in this work.

We now investigate whether training a detector on p2p traffic can mitigate this torrent effect. To this end, we develop training sets with a proportion of p2p traffic which has been reported in Internet study reports [1]. We vary the proportion of p2p traffic in the training set from 40-80% and train NETAD and Maximum Entropy on this training set; TRW-CB and Rate Limiting do not require training and therefore we do not need to evaluate them in the present context. We then evaluate accuracies of the anomaly detectors on the entire dataset (containing all the p2p, malicious and background traffic).

Figure 4 shows the results for training NETAD and Maximum Entropy on different proportions of p2p traffic. It can be clearly seen from Figure 4 that training Maximum Entropy on p2p traffic not only degrades its accuracy but also increases its false positives rate. In case of NETAD, although we observe an increase in detection rate, a 30% increase in false positive rate is induced as we increase the amount of p2p traffic in the training set. This is mainly because p2p clients communicate with each peer on different ports and therefore it is not possible to define an effective filtering rule for NETAD or derive a robust baseline distribution for Maximum Entropy. Hence we conclude that training these anomaly detectors on p2p traffic does not mitigate the torrent effect mainly because contemporary detectors are not designed to filter or incorporate the peculiarities of p2p protocols and clients.

5 Mitigating the Torrent Effect

Based on the empirical accuracy results of the last section, in this section we discuss how can we make an anomaly detector resilient to p2p traffic. While the *right* method to make an anomaly detector resilient to p2p traffic is to avoid detection features which overlap between malicious and p2p traffic, in this section we only discuss an ad hoc method that can be used to make existing IDSs work with p2p traffic. In the following section, we will discuss how future anomaly detectors can inherently cater for p2p traffic in their design philosophy.

5.1 Can a Pragmatic Solution Be Designed to Make an Anomaly Detector Insensitive to p2p Traffic?

Our evaluations in Section 4 show that the torrent effect is mainly caused by initiation of a large number of connections by p2p applications and failed connection attempts in those connections. This behavior of p2p applications is a result of: 1) lack of a central repository in p2p networks to maintain up-to-date information of available peers; and 2) ensuring robustness in p2p networks even with high churn rates. While these key design features of p2p networks can be achieved in a less aggressive manner, p2p applications perform unrelenting attempts to establish connections to thwart techniques to curb p2p connections. The means used to achieve these design goals of p2p networks result in an overlap with malicious behavior.

Since p2p protocols are unlikely to change their behavior in the near-term, and as an IDS designer cannot assume any control over these applications' behaviors,

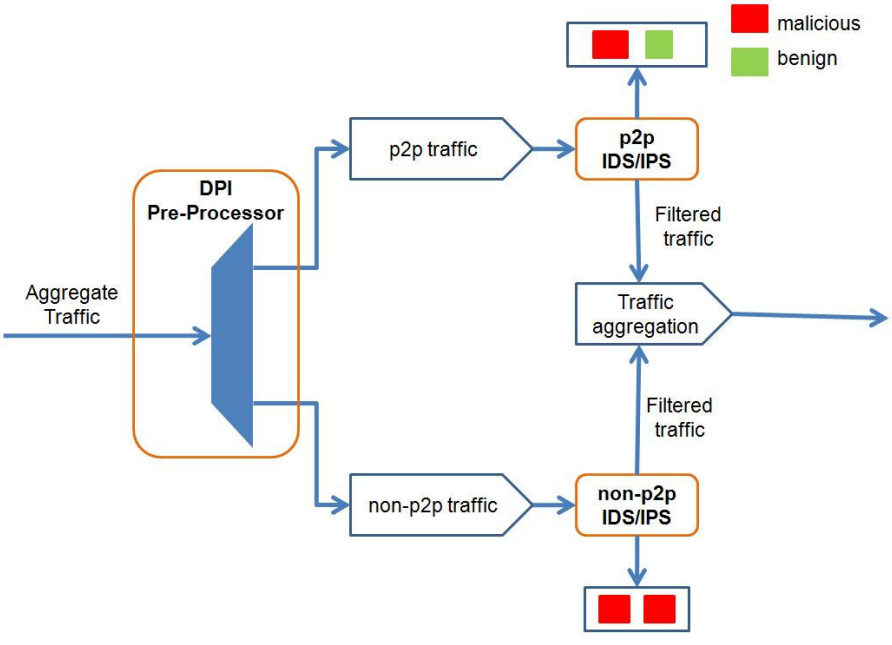


Fig. 5. Mitigating the torrent effect: An IDS with a p2p traffic classification based pre-processor

a simple solution to mitigate the torrent effect is to filter p2p traffic at the input of an anomaly detector using a p2p traffic classifier. Filtering of p2p traffic will result in segregation of non-p2p and p2p traffic as shown in Figure 5. Such a pre-processing filter can be followed by the IDS logic which, in the present context, will only operate on the non-p2p traffic; anomaly detection on the segregated p2p traffic will be discussed in the following section. Since contemporary IDSs are designed to work with non-p2p traffic, detection in the segregated non-p2p traffic will be performed on the unique and non-overlapping characteristics of malicious traffic, thereby yielding good accuracies. This p2p traffic classification based solution has an additional advantage that it requires no changes to be made to existing IDSs. Consequently, at the cost of higher complexity, this generic p2p traffic classification based pre-processor can be integrated into *any* anomaly detector.

There are two problems with this p2p traffic filtering solution: 1) An IDS' accuracy in this design is closely tied to the accuracy of the p2p traffic classifier, i.e., if the p2p traffic classifier can classify p2p traffic accurately, then anomaly detection accuracy will improve, and vice versa; 2) Attacks embedded within p2p traffic will not be detected. The rest of this section address the first point, while the second point is deferred to the next section. In particular, the next subsection answers the following question: Can existing public p2p traffic classification solutions mitigate the torrent effect?

Table 3. Mitigating P2P Effect Using P2P Traffic Classifiers Based Traffic Filtering (DR= Detection Rate; FP= False Positive; KPC= Karagiannis' Payload Classifier)

	Rate Limiting		**TRW-CB**		**MaxEnt**		**NETAD**	
	DR%	FP%	DR%	FP%	DR%	FP%	DR%	FP%
No filtering	50	45	60	22	62	48	65	25
OpenDPI[23]	56	43	64	12	63	32	70	17
KPC[24]	60	40	70	6	66	17	77	13

Table 4. Evaluation of OpenDPI and KPC on Encrypted P2P Traffic

	Classified as p2p	**Classified as unknown**	**Classified as non-p2p**
OpenDPI	3.8%	96.2%	0%
KPC	64.7%	35.2%	0%

5.2 Can Existing Public p2p Traffic Classifiers Mitigate the Torrent Effect?

The p2p traffic classification problem has been well investigated and signature- and heuristic-based solutions exist. We, however, argue that many existing heuristic-based solutions will also be subject to the overlapping feature limitation.[6] Therefore, it is important to choose approaches which use non-overlapping heuristics. We now evaluate our proposed design on a popular DPI-based technology and on a hybrid scheme (signatures + heuristics).

We perform traffic filtering using OpenDPI [23] (a signature based solution with over 90 signatures) and Karagiannis' Payload Classifier(KPC) [24] (a hybrid solution with over 59 signatures); we refer interested readers to the original papers for the details of OpenDPI and KPC. The results of evaluation of the four anomaly detectors on filtered traffic are shown in Table 3. Table 3 shows that KPC (unknown: 35.2%) provides remarkably better accuracy than OpenDPI (unknown: 96.2%), mainly because OpenDPI is unable to detect any encrypted p2p traffic. It can be clearly seen by comparing Table 3 and Table 4 that the improvements in anomaly detectors' accuracies are dependent on the traffic classifier's accuracy. One of the limiting factors in the accuracy of the traffic classifiers is encrypted traffic.

We note from Table 3 that the current traffic classification accuracies of the DPI solutions are inadequate to induce a significant improvement in anomaly detection accuracy; detection rates after p2p traffic classification range from 40-70%, while false positives are between 6-40% for different anomaly detectors. Since the accuracies reported in Table 3 are impractical for commercial deployments, we conclude that public p2p traffic classification solutions at present cannot provide acceptable accuracies to induce an effective accuracy improvement in anomaly detection. While many commercial p2p traffic classification

[6] For example, the method in [20] uses failed connections as a feature and should not be used in the present context.

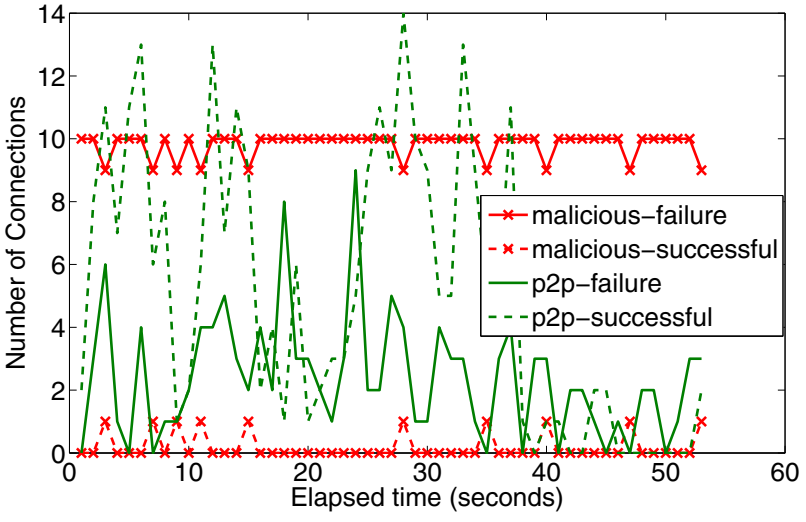


Fig. 6. Connection timeline for p2p and malicious (portscan attack) traffic

solutions are available, to the best of our knowledge, none of the p2p traffic classifiers proposed by the research community have acceptable detection accuracies for encrypted p2p traffic. Therefore, efficient p2p traffic classification remains an open problem and a solution to this problem will benefit the IDS community as well as the traffic engineering community.

Until such a solution is developed, we need to identify non-overlapping (between malicious and p2p) traffic features that an anomaly detector can rely on. As a preliminary, result, Figure 6 shows the connection timeline for the p2p and malicious traffic. It can be seen that the sustained activity of maliciousness is very different from the sporadic p2p traffic activity. Therefore, p2p and malicious traffic can be isolated if a notion of long-term statistics can be introduced during anomaly detection. This is part of our ongoing research.

6 What Are the Open Problems in Designing Future Anomaly Detectors?

The tremendous growth in p2p-based file sharing, VOIP and video streaming traffic has revolutionized the Internet traffic characteristics. Our evaluations showed that this change in traffic characteristics cannot be characterized by existing anomaly detectors which rely on traffic features (e.g., rate, connection failures, ports, etc.) that largely overlap with p2p traffic behavior. While we proposed an adhoc solution which allows existing IDSs to work effectively, a question remains open regarding the scalability of this solution to future Internet traffic. Recent projections of future attacks show that some of the greatest threats in the future will be originating from file sharing networks [28]. In such

a threat landscape, a p2p traffic classification based solution will simply allow all malicious activities embedded within p2p traffic to go undetected.

While detection of malware delivered using p2p applications does not fall under the scope of traffic anomaly detection, attacks originating from p2p networks should be detected using these IDSs. One such attacks has already been proposed in [27] where Naoumov and Ross designed a DDoS engine for flooding a target using the indexing and routing layers in a p2p systems. Similarly, IDSs should be able to detect the exploits targeted at vulnerabilities which are a product of the change to firewall rules for p2p traffic [29]. Finally, it is highly desirable to detect the C&C channels of bots which also use p2p communication [30].

Given the premise that p2p traffic is here to stay, our work demonstrates the need to rethink the classical anomaly detection design philosophy with a focus on performing anomaly detection in the presence p2p traffic. We argue that p2p traffic classification will play a fundamental role in future IDSs as it will facilitate detection of both the p2p and the non-p2p traffic anomalies, as shown in Figure 5. In our proposed design, traditional non-p2p network attacks will be detected using existing anomaly detectors, while an additional IDS that specializes at detecting attacks within p2p traffic will also be deployed.

Design of a p2p-specialized IDS is still an open research problem that is part of our ongoing research and that we also expect our peers to follow-up on. We have made our dataset publicly available for performance benchmarking of such future IDSs and p2p traffic classifiers.

Acknowledgments. We thank Dr. Hyun-chul Kim for providing Karagiannis' Payload Classifier.

References

1. Ipoque Internet Study Report 2008/2009, `http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009`
2. Maier, G., Feldmann, A., Paxson, V., Allman, M.: On Dominant Characteristics of Residential Broadband Internet Traffic. In: IMC (2009)
3. Erman, J., Gerber, A., Hajiaghayi, M.T., Pei, D., Spatscheck, O.: Network-Aware Forward Caching. In: WWW (2009)
4. Labovitz, C., McPherson, D., Iekel-Johnson, S.: 2009 Internet Observatory Report. In: NANGO: NANGO47 (2009)
5. Li, Z., Goyal, A., Chen, Y., Kuzmanovic, A.: Measurement and Diagnosis of Address Misconfigured P2P Traffic. In: IEEE INFOCOM (2010)
6. Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast Portscan Detection Using Sequential Hypothesis Testing. In: IEEE Symposium on Security and Privacy (2004)
7. Schechter, S.E., Jung, J., Berger, W.: Fast Detection of Scanning Worm Infections. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 59–81. Springer, Heidelberg (2004)
8. Williamson, M.M.: Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In: ACSAC (2002)

9. Twycross, J., Williamson, M.M.: Implementing and Testing a Virus Throttle. In: Usenix Security (2003)
10. Gu, Y., McCullum, A., Towsley, D.: Detecting Anomalies in Network Traffic Using Maximum Entropy Estimation. In: ACM IMC (2005)
11. Mahoney, M.V.: Network Traffic Anomaly Detection Based on Packet Bytes. In: ACM Symposium on Applied Computing (2003)
12. Next-Generation Intrusion Detection Expert System (NIDES), http://www.csl.sri.com/projects/nides/
13. Weaver, N., Staniford, S., Paxson, V.: Very Fast Containment of Scanning Worms. In: Usenix Security (2004)
14. Lakhina, A., Crovella, M., Diot, C.: Diagnosing Network-wide Traffic Anomalies. In: ACM SIGCOMM (2004)
15. Lakhina, A., Crovella, M., Diot, C.: Mining Anomalies Using Traffic Feature Distributions. In: ACM SIGCOMM (2005)
16. Patcha, A., Park, J.: An Overview of Anomaly Detection Techniques: Existing Solutions and Latest Technological Trends. Elsevier Computer Networks (2007)
17. DARPA Intrusion Detection Data Sets, http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html
18. LBNL/ICSI Enterprise Tracing Project, http://www.icir.org/enterprise-tracing/download.html
19. Endpoint Dataset, http://wisnet.seecs.edu.pk/projects/ENS/DataSets.html
20. Collins, M., Reiter, M.: Finding Peer-to-Peer File-Sharing Using Coarse Network Behaviors. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 1–17. Springer, Heidelberg (2006)
21. Bartlett, G., Heidemann, J., Papadopoulos, C.: Inherent Behaviors for On-line Detection of Peer-to-Peer File Sharing. In: Proceedings of the 10th IEEE Global Internet (2007)
22. Liu, Y., Guo, Y., Liang, C.: A Survey on Peer-to-Peer Video Streaming Systems. In: Peer-to-peer Networking and Applications (2008)
23. OpenDPI, Ipoque's DPI software's Open Source Version, http://www.opendpi.org/
24. Karagiannis, T., Broido, A., Brownlee, N., Claffy, K.C., Faloutsos, M.: Is P2P Dying or Just Hiding? In: IEEE Globecom (2004)
25. Sun, X., Torres, R., Rao, S.: DDoS Attacks by Subverting Membership Management in P2P Systems. In: 3rd IEEE Workshop on Secure Network Protocols (2007)
26. Athanasopoulos, E., Anagnostakis, K.G., Markatos, E.P.: Misusing Unstructured P2P Systems to Perform DoS Attacks: The Network That Never Forgets. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 130–145. Springer, Heidelberg (2006)
27. Naoumov, N., Ross, K.: Exploiting P2P Systems for DDoS Attacks. In: INFOSCALE (2006)
28. 2010 Cyberthreat Forecast from Kaspersky Lab, http://usa.kaspersky.com/about-us/news-press-releases.php?smnr_id=900000322
29. Chien, E.: Malicious Threats of Peer-to-Peer Networking. Whitepaper, Symantec Security Response (2008)
30. McAfee Labs, Threat Predictions (2010), http://www.mcafee.com/us/local_content/white_papers/7985rpt_labs_threat_predict_1209_v2.pdf

31. Arbor Peakflow: IP Traffic Flow Monitoring System, http://www.arbornetworks.com/index.php?option=com_content&task=view&id=1465&Itemid=692
32. Allot Service Protector, DDoS Protection, http://www.allot.com/Service_Protector.html#products
33. Sandvine: Network Protection, http://www.sandvine.com/products/network_protection.asp
34. Ipoque Press Release: P2P Raid in Germany Shows Little Effect, http://www.ipoque.com/news-and-events/news/pressemitteilung-ipoque-210606.html
35. Ashfaq, A.B., Robert, M.J., Mumtaz, A., Ali, M.Q., Sajjad, A., Khayam, S.A.: A Comparative Analysis of Anomaly Detectors under Portscan Attacks. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 351–371. Springer, Heidelberg (2008)
36. Javed, M., Ashfaq, A.B., Shafiq, M.Z., Khayam, S.A.: On the Inefficient Use of Entropy for Anomaly Detection. In: RAID (2009)

A Centralized Monitoring Infrastructure for Improving DNS Security

Manos Antonakakis, David Dagon, Xiapu Luo, Roberto Perdisci, Wenke Lee, and Justin Bellmor

Georgia Institute of Technology, College of Computing,
Atlanta, GA 30332, USA
{manos,dagon,csxpluo,perdisci,wenke}@cc.gatech.edu,
justin@gtisc.gatech.edu

Abstract. Researchers have recently noted (14; 27) the potential of *fast poisoning* attacks against DNS servers, which allows attackers to easily manipulate records in open recursive DNS resolvers. A vendor-wide upgrade mitigated but did not eliminate this attack. Further, existing DNS protection systems, including bailiwick-checking (12) and IDS-style filtration, do not stop this type of DNS poisoning. We therefore propose Anax, a DNS protection system that detects poisoned records *in cache*.

Our system can observe changes in cached DNS records, and applies machine learning to classify these updates as malicious or benign. We describe our classification features and machine learning model selection process while noting that the proposed approach is easily integrated into existing local network protection systems. To evaluate Anax, we studied cache changes in a geographically diverse set of 300,000 open recursive DNS servers (ORDNSs) over an eight month period. Using hand-verified data as ground truth, evaluation of Anax showed a very low false positive rate (0.6% of all new resource records) and a high detection rate (91.9%).

Keywords: DNS Poisoning, Attack Detection, Local Network Protection.

1 Introduction

The Domain Name System, or DNS, maps domain names to IP addresses and other records essential for email, web, and nearly every significant network protocol. DNS security problems in turn affect numerous other services and critical resources. Recently, the security community has identified *fast poisoning* techniques that allow the trivial corruption of DNS records (23; 14). A poisoning attack allows an adversary to manipulate resolution caches, usually through a "blind" off-path guessing of the transaction components used for DNS message integrity.

Several secure DNS protocols have been proposed, including DNSSEC (6; 7) and DNSCurve (9). DNSCurve provide link-level security while DNSSEC provide object-based security of DNS messages using cryptographic means. However, the deployment of DNSSEC has proven slow (26), and many hosts have on-path hardware that interferes with DNSSEC's larger packet sizes (8).

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 18–37, 2010.

The delay in deploying secure DNS motivates the need for local networks to protect their recursive DNS resolution infrastructure. Traditional solutions such as IDS and packet-inspection tools provide limited protections against some classes of attacks, but do not detect DNS poisonings. Indeed, poisoning attacks generally use valid, "RFC-compliant" DNS messages that contain *misleading* answers (e.g., associating a domain with the wrong IP address or nameserver—one under the control of an attacker).

For this reason, DNS security systems are generally concerned with records *in cache* (or in the resolver), as opposed to *in flight* (or on the wire). In this work, we focus on *in cache* detection of DNS poisoning for similar reasons:

1. The in-line inspection of DNS traffic can introduce latency. Some protocols are tolerant of this delay, but for DNS, even adding a few tens of milliseconds delay can have detrimental impact on other services (e.g., VoIP, DNSBL validation, etc.). In extreme cases, adding such delays can result in SERVFAIL responses.
2. Several tools already detect classes of DNS attacks, such as packet format violations (e.g., name pointer loops (4)). These attacks are orthogonal to DNS poisoning, and must be done *on the wire data*, as opposed to the cached data.
3. Some DNS attacks, such as out-of-bailiwick record injection (35), are already rejected by the DNS resolvers themselves. Such attacks are technically DNS poisoning, but have been addressed by RFC 2181 (19) (and related policies) and are routinely dropped by recursive servers. (This is known as "answer validation" in most DNS resolvers (12).) The DNS poisoning attacks we consider are in the newer family of *fast poisoning* or "Kaminsky-class" attacks, which evade these forms of basic RFC 2181 trustworthiness checks. Note that the answer validation phase is usually opaque in a DNS resolver, and server logging of rejected records is often infeasible, mainly due to system performance and volume of the logs.

For these reasons, we focus on the detection of DNS poisoning that is found *in cache*, in order to identify attacks that have evaded all existing layers of protection. To detect DNS poisoning that has evaded all other layers of filtration, we need access to large, busy recursive servers. In practice, such access is difficult to obtain, because of the operational risk it poses to a critical network component, and because of potential privacy concerns in witnessing stub traffic. We therefore decided to use data obtained from the inspection of open recursive caches run by third parties on the Internet. Open recursive resolvers (16) generally permit the inspection of their caches. Since we can successfully detect poisonous Resource Records (RR) in Internet scale measurements, we will be able to do the same when we inspect a less diverse set of recursive DNS servers, e.g., those in a single organization.

We select 300,000 open recursive servers, in order to obtain a diversity of DNS resolvers based on geography, network size, and organizational type (e.g., corporate vs university networks). The network properties of these hosts are discussed in Section 3. Using this data source, we designed and evaluated a large-scale, centralized poisoning detection system called *Anax*. Our implementation of Anax provides a scalable, centralized view of DNS poisoning. Further, it works in an automated manner with minimal human intervention. Anax is able to perform these measurements without being on the same network path as the attacker and victim. During our experiments, Anax was able to

successfully detect 319 unique poisoned resource records (RRs) that were subsequently manually verified as DNS poisoning attacks. In addition, because Anax works on arbitrary DNS caches, it can also protect local networks against poisoning even when the local resolver is not open recursive.

Anax relies on a fundamental observation about DNS. Despite being dynamic, DNS records generally direct users to a known, usually stable set of `NS` records. Poisonings on the other hand, generally redirect victims to new, different IP addresses often set up for furtive, short-lived harvesting of information (such as banking credentials, credit card numbers and email passwords). We therefore created detection heuristics that note the statistical DNS properties of answers. Our analysis shows that our features are stable even against significant changes in legitimate DNS hosting.

We operated the Anax poisoning detection system for several months, resulting in a database of tens of millions of DNS answer records. Using extensive classification filters and heuristics we can reliably label the majority of the IPs in recorded RRs. Using manual effort we verified by hand and labeled the remaining 1,264 unique IPs address record as "legitimate" and "poisonous". This labeled data set was then used to train and test our detection module, as described in Section 3. The evaluation of Anax based upon real world data proved so promising that it makes our system an efficient real-time poisoning detection system.

The remainder of this paper is organized as follows. Section 2 provides in-depth technical details of poisoning attacks and related work. Section 3 presents the detection methodology that Anax utilizes. Section 4 details our experiments with Anax, including validation and labeling steps of Anax's dataset. In Section 5 we elaborate on the details of the detection heuristics that Anax uses and present the detection results based on our real-world data analysis. Finally, we conclude in Section 6.

2 Background and Related Work

This section offers a brief overview of the Domain Name System (DNS), addressing aspects relevant to poisoning and detection. Readers familiar with DNS may skip over this section. Further background on DNS can be found in (37).

2.1 Background on DNS Poisoning

DNS provides a distributed database of domain names organized as a tree structure. A domain name is a node in the tree and is labeled with the minimum path used to reach the node from the root. When expressed as a fully qualified domain name, each node is a label separated by period. A *zone* is a collection of nodes under a common parent. Such collections form a subtree, the top of which is called the *start of authority*. Authority DNS servers answer queries about nodes in their zones, and generally provide answers about mappings of *leaf nodes* (or terminus nodes), or a referral to another sibling authority when sub-zones have been delegated to another authority server. The answers from such authority servers are recorded by recursive DNS servers for caching on local networks.

Although DNS poisoning could occur between the stub and forwarder (step one), or the forwarder and resolver (step three), we are primarily concerned with attacks on the

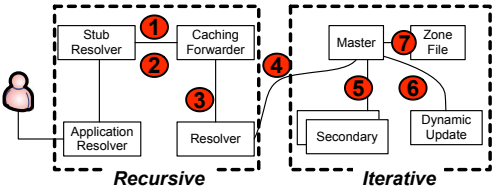


Fig. 1. An overview of DNS resolution, and risks posed at each phase of the resolution path. DNS poisoning is most commonly concerned only with risks experienced on step four, the communication between resolvers and authorities.

path between the resolver and authority (step four in Figure 1). This path is by necessity exposed to the Internet. Since DNS responses are (with noted exceptions (36)) usually a single UDP packet, attackers can send large numbers of spoofed, malicious answers that are "off-path". By "off-path" we mean that an attacker can spoof a UDP packet, claiming to be the authority for a zone from any point on the Internet. Witnessing such poisoning attacks requires the observer to be "on-path" (e.g., as a transit provider or below/above the resolver). If one is not "on-path", it is often difficult to observe such DNS attacks (15).

The basic properties of traditional and Kaminsky-class DNS poisoning attacks have been extensively studied (15; 16; 27). The Kaminsky-class of DNS attack greatly speeds up traditional DNS poisoning attacks that have historically been done by changing stub DNS settings (16), shown in step two of Figure 1. This increase in the attack speed due to Kamisnky class of poisoning can be achieved by repeatedly attempting to poison "new" nonce names in a zone of interest. Even a bandwidth limited attacker will eventually win the packet race for one of the nonce child names (14), allowing for replacement of the `NS-type` of record in cache. Recent industry studies have noted that DNS manipulations are not only used for phishing, but commonly used for "click-fraud" and by spammers to drive traffic to malicious sites (32), as well.

2.2 Related Work

Our work combines ideas from two areas of literature: DNS cache poisoning detection and Internet-wide DNS-based measurement. While Anax is the first system to detect Kaminsky-style DNS cache poisoning, it owes much to previous related research.

DNS cache poisoning is not a new phenomenon. Cache poisoning has been a known vulnerability in DNS since at least 1993 (33), and has seen a resurgence issue in 1997 (35), 2002 (10), 2007 (25), and 2008 (23). Despite many years of research in eliminating cache poisoning, the latest attack was judged serious enough to warrant multi-vendor coordinated patching (3).

Several vulnerability assessment tools and technologies allow the discovery of DNS vulnerabilities often caused by misconfiguration. Nessus (1) and specific DNS related tools such as DNSStuff (17) and PorkBind (11), detect DNS servers vulnerable to specific cache poisoning attacks. In contrast, Anax detects actual cache poisoning instead of vulnerabilities.

No available tool exists to detect actual *in-cache* poisoning. DoX (41) would use a peer-to-peer network to detect cache poisoning, but it has never been tested in practice nor deployed on the Internet, and this system would require a significant infrastructure and the cooperation of other DoX nodes to be effective. In contrast to DoX, Anax is a centralized system, does not require any external cooperation, and has been tested on real world network scenarios.

Several solutions, such as DNSSEC (6; 7), DNSCurve (9), 0x20 encoding (15) and WSEC-DNS (27), have been proposed to eliminate cache poisoning vulnerabilities entirely. While these solutions would reduce or eliminate cache poisoning, they require explicit or implicit changes to the DNS protocol, are not widely deployed, or are not likely to find wide-spread adoption in the short term (maybe except DNSSEC).

Internet-wide measurement via DNS has been previously used to estimate delay between two arbitrary hosts in King (21). Anax's goal is not to measure distances between arbitrary hosts, as King does, but to collect IP information about a set of "domain names of interest" (detailed in Section 3.2) that King does not. Internet-wide DNS poisoning scans have been performed by The Measurement Factory (20), but these scans only investigate parent zone poisoning, to which very few name servers are vulnerable, while Anax can detect Kaminsky-class attacks, to which many currently deployed servers are vulnerable. Anax is also able to detect cache poisoning targeted at a specific resolver or set of resolvers. Wendlandt et al. (38), proposed "Perspectives", a system that uses multiple hosts to verify a server's public key. Our system has a similar scanning methodology but the scope of the two systems is orthogonal; Anax deals with DNS RR validation within cache, while "Perspectives" reactively validates public keys.

Finally, we note that our work has a superficial similarity to the Notos domain reputation system (5). Notos, created by many of the same authors of this work, uses machine learning to assign a reputation score to unknown domains according to given trained categories (e.g., spam-related domains, botnet domains). In contrast, the present study uses a very limited set of features to identify poisonous DNS records. While Notos allows one to identify groups of similar domains, Anax lets one judge the integrity of selected *in-cache* records.

3 Methodology

In this section we describe the methods that Anax uses to detect cache poisoning. We start with a discussion of the features inherent to cache poisoning attacks, in particular how poisoning attacks may be detected by observing changes in records cached by open-recursive DNS server (ORDNS).

Figure 2 shows the overview of the Anax poisoning detection system. In step one the scanning engine sends to the scanning host a list of domain names and ORDNS servers. The raw DNS answers from scanning (step two) are stored in the raw DNS data collector. A one-time training step labels and verifies a portion of these records (step three). After manually labeling the dataset, we send it to the detection engine for modeling (step four). The resulting models around the benign and poisonous classes of RRs will be stored in the Anax DB. At this point the system can be directly utilized (step five) to classify new unknown RRs in DNS answers as they arrive from the scanning

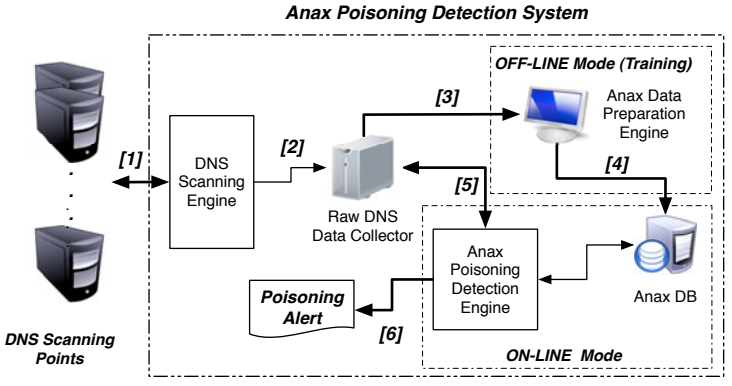


Fig. 2. Anax Poisoning Detection System

points to the raw DNS data collector. Then, Anax can be switched to an on-line mode, and detect new poisonous records using step six.

3.1 Abnormality in DNS Answers Due to Cache Poisoning

Kaminsky-class attacks have made cache poisoning even easier, especially against unpatched servers or servers that cannot take advantage of full source port randomization due to network configurations like NAT. As noted in section 2, poisoning attacks create inherently *local* impacts, making it hard to observe once you are "off-path" of the resolver.

The consensus of answers observed in the wild can be used to validate the resource records (RRs) presented as valid answers. In practice, there are several nuances to this simple approach. DNS can be used for load balancing, localizing content, and to monetize typographical errors, so query results often vary, even without malicious manipulation. To avoid effects of load balancing and content localization, it is necessary to obtain consensus results based on *network* and *geographic diversity*.

An ORDNS that has been the victim of a cache poisoning attack, will answer "on-path" queries using somehow different IP(s) in these RRs (Table 1, lower sub-table) or NS(s) that cannot be correlated with the domain name we try to resolve (Table 1, upper sub-table). Usually these IP(s) point to a different, attacker controlled server. The answer for the poisoned record should inevitably contain at least a single different IP than the IPs found in legitimate RRs for the same domain name. As noted in Section 2, the only possible way to observe this variation in answers is to be "on-path" with the ORDNS. In other words, one needs to be able to directly query the resolver for the poisoned RR. Since we did not have access to customer transit data for this study, we generated such data by utilizing two DNS scanning points: one located in California and one located in Ottawa. Using these two scanning points we probed a large, geographically and network diverse set of open recursive DNS servers, as discussed below.

Table 1. Poisoning cases observed by Anax. In the upper part of the table we can see NS replacements observed in `NS-type` RRs. In the lower table we can see IPs in `A-type` RRs that were manually labeled as poisoning cases. With the ORDNS column we provide the type of ORDNS software from the poisoned resolver using the fpdns tool.

Domain Name	NS	CC	Date	ORDNS
amazon.com	hu-bud02a-dhcp09-main.chello.hu	HU	2009-07-26	Cisco CNR
americanexpress.com	c.exam-ple.com	PA	2009-03-20	BIND 9.2.3
americanexpress.com	d.exam-ple.com	PA	2009-05-05	Win DNS NT4
bankofamerica.com	209.59.194.246	US	2009-06-18	Win DNS 2003
bankofamerica.com	209.59.195.246	US	2009-06-18	Win DNS 2003

Domain Name	IPs	CC	Owner	ORDNS
americanexpress.com	189.38.88.129	BR	CYBERWEB	BIND 9.2.3
google.com	85.10.198.253	DE	HETZNER-AS	Win DNS 2000
visa.com	61.207.9.4	JP	OCN NTT	BIND 9.2.0
update.microsoft.com	205.178.145.65	US	Net. Sol.	No Match
google.com	65.98.8.192	US	FORTRESSITX	QuickDNS

To identify Kaminsky-class attacks (`NS-type` record replacements) and simple DNS poisonings (`A-type` record manipulations), Anax relies on an inherent feature of DNS poisoning: namely, that the poisoned ORDNS will report cached RRs that are "abnormal" with respect to zone and the IP address space. We define as an abnormal the RR with an IP that **should not** reside nor can be linked in any way with the poisoned zone's **"network provisioning"** — a network that can be associated with the zone's operator or a major Content Delivery Network (CDN). For example, a poisonous NS record for amazon.com will point hosts to an authoritative name server (ANS) outside of Amazon's typical DNS provisioning address space. In other words, the IP address of the attacker controlled ANS along with the IP address in the poisoned `A-type` records, cannot be linked with Amazon's IP address space or even worse it might be in dynamic address space. This variation in the RRs can be measured externally as long as we can be "on-path" with the ORDNS.

3.2 Probes and Measurements

Anax's poisoning detection works in three discrete phases: *preparation, measurement*, and *analysis*. The preparation phase consists of collecting IP addresses of open-recursive DNS servers located throughout the world, determining which domains could be likely targets of poisoning attacks, and probing open-recursive servers for poisoning detection (DNS Scanning Engine, Figure 2).

During the *measurement* phase, Anax's scanning engine performs a series of queries while recording matching answers. All the resulting raw DNS traffic is placed in a fully indexed database (Raw DNS Data Collector, Figure 2). Finally, in the analysis phase (Data Labeling and Detection Engine, Figure 2), Anax performs a series of checks on the recorded RRs from all scanned open-recursive servers. Anax will be able to assign a label for each unique RR of a given zone, and decide its legitimacy. The preparation and measurement phases are described below; the analysis phase is described in Section 5.

Preparation. The preparation phase of Anax is composed of three parts: the *gathering* of ORDNS servers, the *identification* of domains likely to be poisoned, and the *probing* of each ORDNS server for poisoning detection. ORDNS servers are gathered using the method proposed by Dagon, et al. in (16). Using this method, we were able to obtain 8,274,341 open-recursive DNS servers distributed throughout the world. Anax also periodically re-checks DNS resolvers to ensure they continue to behave as open-recursive servers. It is very expensive to regularly probe all discovered ORDNS, therefore we sampled a smaller but geographically diverse set of 300,000 ORDNSs. We made hundreds of thousands of DNS queries to a large, geographically and network diverse set of these 300,000 ORDNSs for 131 zones of interest. A small glimpse of the overall ORDNS diversity from our scanning list with regard to the country code (CC), the autonomous systems (AS) and CIDR block can be found in Figure 4.

Since traditional cache poisoning attacks only affect DNS cache entries for a specific domain, poisoning may only be checked on a per-domain basis. To create a list of domains that are likely to be attacked, we combined the top 100 worldwide websites as ranked by Alexa with the world's top 100 e-business websites, yielding 131 unique domains. These 131 domains are globally distributed, focus on a variety of industries, and all have very high visitor counts. To the best of our knowledge, none of these domains are used for malicious operation, and theoretically the domain names and IPs from these sites should not be part of any black list. The amount of financial transactions conducted through these sites also makes them very tempting targets for phishing attacks (as noted by several on-line phishing analysis resources (34)), that potentially could be staged via DNS poisoning. We refer to this list of 131 domains as the "domains of interest".

Measurement. Anax uses repeated queries to discover IP address records for the domains of interest. Using the following scanning protocol, Anax maintains `A-type` record information and `NS-type` record information for the domains of interest.

Anax's scanning points issue a series of typical DNS queries like the one presented in Figure 3. These scan points use such queries in order to capture the *on-path* behavior of the ORDNS. A scan point always makes four types of queries to an ORDNS for each of the domains of interest. The type of queries are `A`, `NS`, `MX` and `AAAA`. The main

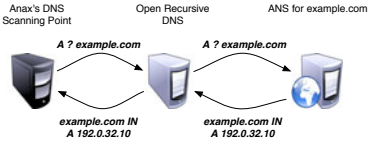


Fig. 3. A typical `A-type` query for example.com to an open-recursive server (ORDNS). In this case the ORDNS's cache is empty, and the ORDNS needs to ask the authoritative name server (ANS) of example.com in order to find the IP that is currently "mapped" to the domain name.

CC	#ORDNS	#ASs	#CIDRs
US	116213	3785	14340
CN	34778	90	2574
JP	20147	329	1760
NL	17651	172	483
FR	16261	164	482
KR	14822	326	1316
IT	12824	204	569
GB	9587	414	952
DE	9441	408	818
SE	9119	113	355

Fig. 4. A summary of the diverse ORDNS scanning targets

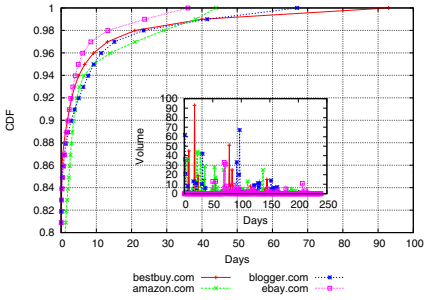


Fig. 5. IP discovery trend in Anax for zones that use CDN networks or are network diverse

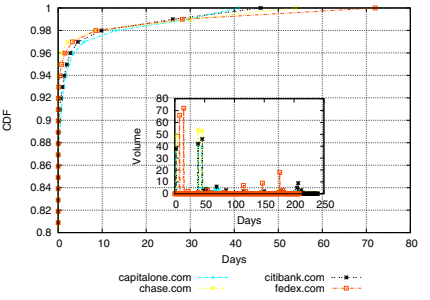


Fig. 6. IP discovery trend in Anax for network stable zones

reason for selecting these different types of queries is to discover as many different RRs as possible for each zone without requiring access to the zone itself.

Let us assume that d is a domain of interest. The complete probing protocol we use is the following querying sequence: the **first** query is an `A-type` and for the domain $d_{control}$, which we own and for which we operate the only "legitimate" authority name server (ANS). The `A-type` of record for this domain contains a single IP that we never change. Using this simple technique we can check if the ORDNS server we probe is acting as a real open-recursive, is mis-configured or provides a DNS-tunneling service. The **second** query is an `A-type` and for the domain d. These queries provide Anax with the current IP address of the probed domain d. The **third** is an `A-type` query for $random\_value.d$. Since the random nonce ($random\_value$) does not exist, and the zone does not use wild-card entries, this query ensures that the remote ORDNS always reaches the authority name server (ANS) of d. This results in an answer of NXDOMAIN. We are certain about this since we can trivially verify that none of the zones of interest are wild-carded in the 2^{nd} level domain (2LD). The **fourth** query is an `NS-type` query for d, from which Anax discovers the current IP and domain name information about the authority name servers for the domain name d. Some CDN enabled zones (e.g., bestbuy.com) tend to have their authority name servers operated by the CDN network. We want to capture this diversity in our dataset.The **fifth** query is an `MX-type` query for d, which provides us with the current email servers for the domain name d. Typically this IP location is managed by the same owner of the domain name d. Some zones, however, simply outsource their e-mail services (e.g., AT&T and MessageLabs). The **sixth** is an `AAAA-type` query for the domain name d, based on which we can record again the start of authority record (`SOA-type`) for the domain d.

As noted above, we probe using this sequence of queries so we can obtain key characteristics about the open-recursive server in a single scan event: IP and nameserver information for each domain of interest. We discover as many IP to domain name mappings for each zone as possible due to query type variation (`A, NS, MX, AAAA`). In Figures 5 and 6 we can see a sample of the two observed IP address discovery growth trends that we observe with Anax's scanning engine. Figure 5 shows that domain names that exhibit significant network diversity for IP address present in legitimate `A-type`

answers (e.g., blogger.com, amazon.com) or utilize CDN networks (e.g., bestbuy.com uses akamai.net), Anax needs more time to identify all possible IPs addresses. In this case Anax will identify 95% of address records for these network diverse zones in 8-12 days while at the same time continue discovering new IP addresses months after the start of scanning. In Figure 6 domain names with a more stable network profile utilize significantly fewer IP addresses over time. In this case Anax takes less time (95% of address records will be discovered in 2-3 days) to discover almost all of them.

4 Dataset Evaluation

Using the Anax infrastructure described in Section 3, we periodically made a large number of DNS queries to a set of 300,000 ORDNSs for the 131 zones of interest. The raw DNS collector holds the DNS data generated by Anax's scanning engines. The scanning points periodically synchronize their data to this server for analysis. When the system is in on-line mode (Figure 2; step five) the data can be instantly classified as it arrives in the raw DNS collector. Potentially, the detection engine could be placed directly at the scanning points and classify new RRs on-the-fly.

We evaluated the system in its off-line mode since it was necessary for us to carefully obtain ground truth for our classification process. We used part of the captured traffic to evaluate our detection algorithm. For training our detection system, we used 23 million DNS answers recorded between January 2009 ant the end of February 2009. To create the testing dataset, we used 57 million DNS answers recorded between March 2009 and August 2009.

The raw DNS data gathered by Anax holds all possible observations made about the resource records (RRs) in the received answers for all zones of interest. Our dataset provides "evidence"—that is the RRs ("Domain Name to IP" and "Domain Name to NS server") returned by the ORDNS. A portion of the unique RRs present in these datasets were manually classified to provide the ground truth for our study. At this point we should note that Anax is able to classify `A-type` resource records or "Domain Name to IP" mappings. The collection of the `NS-type` records (or "Domain Name to Name Server" mappings) helped us in the manual classification process and forensic analysis of the hand verified poisoning cases.

4.1 Dataset Labeling

We constructed our limited whitelist by selecting 23 "major" recursive DNS servers across the US. Using a one-time probe against these open-recursive servers, we obtained all address records for the 131 zones of interest. We hand verified that each address found in the returned answers was indeed part of the legitimate domain resolution. After mapping the returned IPs to the corresponding Classless Inter-Domain Routing (CIDR) block, we used this newly created set of CIDRs as our **only** CIDR based whitelist.

During our eight month scanning period, while most of the answers were deemed legitimate, not all illegitimate answers were necessarily poisonous. DNS misconfiguration is a common phenomenon, and sometimes resembles malicious behavior (39; 16). To account for this, we created several labels for a range of responses: Legitimate

Response, CDN, Misconfiguration, NXDOMAIN rewriting, DNS Proxy, and Poisoning, as described below:

Legitimate Response: Legitimate responses indicate a properly functioning ORDNS server returning correct results. The resolutions and authoritative name servers for the list of domain names of interest all point to machines in the same autonomous system among open-recursive servers in the same geographic region, and match prior, verified answers. Our initial whitelist is a small subset of this category.

Content Delivery Networks (CDN): Many zones use DNS to load balance and localize web traffic for popular destinations. This appears where the domain's addresses are assigned to a known content delivery network such as Akamai or Limelight. Network blocks operated by content delivery networks are highly diverse and it is difficult to whitelist all their members. We consulted passive DNS databases (e.g., (22)), to assist on labeling IPs on this category.

Misconfiguration: Some answers showed clear signs of misconfiguration. This is often seen when hosts answer as an authoritative for the root servers or common TLDs such as `.com`, `.net`, or `.org`, or instead return an RFC 1918 or RFC 3330 address. These errant authoritative answers are described in (39) as misconfiguration.

NXDOMAIN rewriting Services: When an IP address was returned for a query that should elicit an NXDOMAIN response, the result was labeled as NXDOMAIN rewriting. These results are not cases of malicious poisoning, and can be detected when an open-recursive DNS server returns an IP address instead of NXDOMAIN for a domain known not to exist. Generally, the resulting IP address points to an advertising portal or a search engine.

DNS proxy: When the ORDNS always provides the same IP address for multiple zone and query types, and at the same time we can identify it as DNS tunnel (2) or a ToTD (31) server, we classify it as DNS proxy. Most tellingly, such resolvers exhibit no IP variations, since they never consult authorities and maintain no cache. Strictly speaking, we do not treat this as DNS poisoning, even though local networks may likely wish to ban the use of DNS proxies.

Poisoning: We **hand-verify** and label as "poisonous" any address returned by an ORDNS that was not owned by the domain name owner, and pointed to a machine under the control of a malicious party. To assist with this labeling, we consulted numerous IP blacklists (30; 24; 13), do-not-route-lists (28), dynamic IP space (29) and passive DNS databases (22). IPs in RRs that pointed to such hosts indicated malicious poisoning of an ORDNS.

5 Detection Model and Results

The poison detection flow in Anax consists of detection modules placed in series to reduce false positives and produce as few false negatives as possible. (As noted below, we arranged these detection modules to place the highest false positive rate first, to maximize the final true detection rate) Figure 7 shows the various steps of the detection flow. RRs not in the Anax DB are forwarded to the CIDR analysis module defined

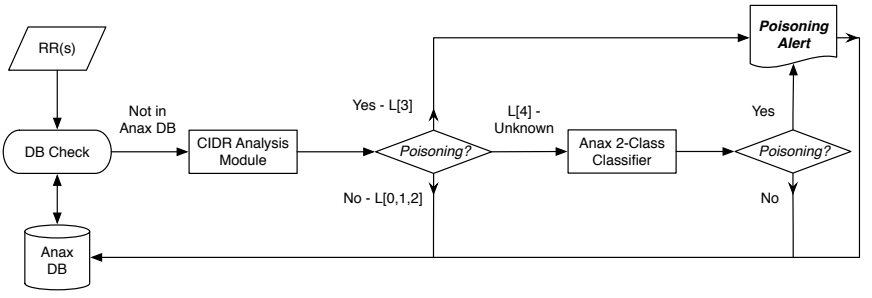


Fig. 7. Poisoning detection flow in Anax

in Section 5.2. The L[0]-L[4] categories (defined in Section 5.1) represent the various ways the CIDR analysis module may classify a new RR. The 2-Class classifier (defined in Section 5.3) handles the unknown RRs from the CIDR analysis module (category L[4]) and flags them as benign or poisonous, based on trained models of benign and poisonous RRs. In the case of poisonings, Anax produces a detailed poisoning report on the RR that caused this alert. The detection flow ends by updating the Anax DB on the analyzed RRs. In the following sections we examine in detail the key modules of Anax's detection flow, namely the CIDR analysis module and Anax's 2-class classifier.

5.1 Categories of Resource Records

In Section 4.1 we identified the type of DNS responses we anticipate to receive. Before we elaborate on the details of the CIDR analysis module and 2-Class classifier we introduce the categories of resource records that the modules can handle. Anax groups RRs in the following five categories in order to clearly define the detection actions that each detection module will enforce. These five categories are:

L[0] - Whitelisted RRs: This category is comprised of address records known to be benign, based on our small CIDR-based whitelist.

L[1] - Misconfiguration & "non-routable" IPs: This category is comprised of RRs with IPs that should be considered as misconfigurations since they point to "non-routable" address space. Although interesting, they are not useful for detecting DNS poisoning and are counted as benign when calculating the final detection rates.

L[2] - NXDomain rewriting & Proxies: This category contains two special cases of `A-type` records. The first category contains address records that are meant to produce NXDOMAIN answers according to our probing protocol but did not. Addresses from such NXDOMAIN rewriting services are of no interest as poison, and are deemed benign. The second category is composed of address records from ORDNS acting as DNS tunneling servers. There is no interest in further analyzing these RRs as poison.

L[3] - Poisonous RRs: This category includes address records that reside in any of the following public lists: do-not-route or peer list (28), dynamic IP address space (PBL) (29), hosts reported to drop malware (XBL) (30) or engage in other malicious activity (24). These RRs will cause the detection algorithm to exit while creating a poisoning alert. The reason why these records will be considered as cases of poisoning attacks is that none of the domains present in our list of "domains of interest" would ever internationally serve malware. Therefore, none of the IPs present in their resource records should ever be in any of these lists.

L[4] - Unknown RRs: This category contains address records from which the CIDR analysis module (defined in Section 5.2) can make no immediate detection decision based on the four previous categories. Finer grained analysis is needed for these RRs. As we will describe in Section 5.3, this can be achieved by computing a six-dimension statistical feature vector.

5.2 CIDR Analysis Module

Address records in RRs that fall into these five categories will initially be used by the CIDR analysis to either create a poisoning report (category L[3]), claim that the RR is not the subject of a poisoning attack (categories L[0]-L[2]) or forward any RR that requires more expensive, fine-grained analysis to the Anax 2-Class classifier (category L[4]). The CIDR analysis module receives RRs, such us `google.com IN 74.125.67.104`, with IP addresses from the monitored zones. Its goal is to make an immediate detection decision about the address record. Based on the categories mentioned in Section 5.1, any IP address within the RR will reside in one of the five categories (L[0]-L[4]).

The primary motivation behind the use of this module is to reduce the overall false positives and to eliminate unnecessary analysis of IPs that fall into the L[0]-L[3] categories. Address records marked as L[0], L[1] and L[2] will cause the detection algorithm to exit without producing a poisoning alert. Simultaneously, the detection algorithm will update the Anax database. For address records that will be placed in the L[3] category by the CIDR analysis module, a poisoning alert will be generated for the corresponding RR. RRs that falls into the L[4] category will be forwarded to the 2-Class classifier, which will make the final detection decision based on statistical models from known benign and poisonous RRs profiles.

5.3 Anax 2-Class Classifier

RRs in category L[4] that cannot be directly checked with our limited white and black listing. Therefore, we use a 2-class K-nearest neighbors (IBK) classifier to make the final detection decision on them. This statistical classifier differentiates between benign and poisonous RRs based on benign and malicious RR profiles built using passive DNS information. Passive DNS data collection is a very common technique that gathers historic DNS resolutions. We use such passive DNS data traces (pDNS) to produce six statistical features for Anax's 2-class classifier.

In order to compute these features, Anax requires a resource record (RR) as an input. An RR of `A-type`, as we already mentioned in previous sections, is composed of a

domain name d and an IP d_{ip}. We define $BGP(d_{ip})$ as the set of all IPs in the same BGP prefix of d_{ip}. Each domain name present in our list of "domains of interest" is composed of two parts: the top level domain or TLD (e.g., .com, .org) and the second level domain or 2LD (e.g., ebay, google). We represent every domain name d in our list as $d_{2ld}.d_{tld}$. Using the same logic, when we query the pDNS against an IP, the pDNS will report back to us a list of domain names that are historically linked with this particular IP. We refer to each returned domain name from the pDNS DB as AD. Each returned domain name can also be represented as $AD = ad_{nld}.....ad_{2ld}.ad_{tld}$, assuming that it is a n^{th} level domain.

The set of all **unique** domain names returned from a passive DNS query on d_{ip} is $APDNS_{d_{ip}} = \bigcup_{k=1..m} AD_k$, where m is the number of unique domain names (AD) that historically can be linked with the d_{ip} in the passive DNS database (22). Also, we define $APDNS_{BGP(d_{ip})} = \bigcup_{k=1..m} AD_k$, where m is the number of unique domain names (AD) that historically can be linked with any IP in the BGP prefix of d_{ip} in the passive DNS DB. Next we define $AD^{3ld.2ld.tld} = ad_{3ld}.ad_{2ld}.ad_{tld}$, $AD^{2ld.tld} = ad_{2ld}.ad_{tld}$ and $AD^{2ld} = ad_{2ld}$.

Now we can define the set $APDNS^{3ld.2ld.tld}_{d_{ip}} = \bigcup_{k=1..m} AD^{3ld.2ld.tld}(k)$ which include all $AD^{3ld.2ld.tld}$ domains (e.g., www.example.com) from all domain names in the set $APDNS_{d_{ip}}$. We also can define the set $APDNS^{2ld.tld}_{d_{ip}} = \bigcup_{k=1..m} AD^{2ld.tld}(k)$ which include all $AD^{2ld.tld}$ domains (e.g., example.com) from all domain names in the set $APDNS_{d_{ip}}$. We define as $APDNS^{2ld}_{d_{ip}} = \bigcup_{k=1..m} AD^{2ld}(k)$ the set of strings which include all AD^{2ld} (e.g., example) from all domain names in the set $APDNS_{d_{ip}}$.

Similarly, we define the two sets $APDNS^{2ld.tld}_{BGP(d_{ip})} = \bigcup_{k=1..m} AD^{2ld.tld}(k)$ and $APDNS^{2ld}_{BGP(d_{ip})} = \bigcup_{k=1..m} AD^{2ld}(k)$ that include all second level domain names ($AD^{2ld.tld}$) and all strings (AD^{2ld}) from all domain names in the set $APDNS_{BGP(d_{ip})}$ respectively.

Finally, we define a list of popular second level domains (2LD) that belong to content delivery networks (CDN) like Akamai, CoralCDN, Limelight and Redcondor. We refer to this list as $ACDN = \bigcup_{k=1..n} cdn_k$, where cdn_k is a distinct **fully qualified** second level domain name (e.g., akamai.net, akamaiedge.net, coralcdn.net). We now elaborate on how we compute the six statistical features based on each newly received resource record:

[Φ_1] - Domain Name Diversity: The number of **unique** domains in the set $APDNS_{d_{ip}}$ that historically have been mapped with the d_{ip} in the RR.

[Φ_2] - 2LD Diversity: The number of **unique** $AD^{2ld.tld}$ present in the set $APDNS^{2ld.tld}_{d_{ip}}$ and have been historically mapped with the d_{ip} in the RR.

[Φ_3] - 3LD Diversity: The number of **unique** $AD^{3ld.2ld.tld}$ present in the set in the set $APDNS^{3ld.2ld.tld}_{d_{ip}}$ and have been historically mapped with the IPs in d_{ip}.

[Φ_4] - Relative BGP CDN Occurrence: The frequency of the $AD^{2ld.tld}$ that historically are present in the set $APDNS_{BGP(d_{ip})}$ and at the same time the $AD^{2ld.tld} \in ACDN$.

[Φ_5] - Relative BGP $\mathbf{d_{2ld}.d_{tld}}$ Occurrence: The frequency of the $d_{2ld}.d_{tld}$ in the set $APDNS^{2ld.tld}_{BGP(d_{ip})}$ that historically have been mapped with any IP present in the set $BGP(d_{ip})$.

[Φ_6] - Relative BGP $\mathbf{d_{2ld}}$ String Occurrence: The frequency of the string d_{2ld} in the set $APDNS^{2ld}_{BGP(d_{ip})}$ that historically have been mapped with any IP present in the $BGP(d_{ip})$.

The statistical features Φ_1, Φ_2 and Φ_3 will provide us with historic DNS information based only on the d_{ip} in the RR. The statistical feature Φ_4 will capture the participation of commonly used CDN second level domains that historically have been mapped with any IP in the same BGP prefix as the d_{ip}. Finally, the statistical features Φ_5 and Φ_6 will capture the participation of all other domain names that point into the same BGP prefix with the d_{ip} and at the same time match with the $2ld.tld$ and the $2ld$ of the domain d.

If the 2-Class classifier labels the IP as poisonous, a poisoning alert will be created for the corresponding RR. Otherwise, it will be marked as benign and it will be added into Anax DB.

5.4 Model Selection and Detection Results

We evaluate the 2-Class classifier in two modes: standalone mode and "in-line" with the CIDR analysis module. In the standalone mode we seed the classifier with any new RRs directly, while in the in-line mode we feed the RRs to the CIDR analysis module and the classifier receives only RRs that belong solely to the L[4] (unknown) category. We evaluated our modules with this process to better justify our decision of assembling the detection flow the way we did. It is straightforward, from an efficiency-minded point of view, that placing the CIDR module in-front of the classifier should lessen the workload on the classifier (since IPs labeled L[0] - L[3] need no further processing). The question we try to answer in this section is the following: will the classifier perform better in in-line or in standalone mode?

We start by carrying out the model selection, a very common technique from the machine learning community. Model selection is used in order to select the optimal machine learning method for solving a given classification problem (18). We select one classifier for each major family of commonly used classifiers:

I. **Simple Logistic Regression - SLR;** a classifier for building linear logistic regression models.
II. **K-nearest neighbors classifier - IBK;** a "lazy" K-nearest neighbors classifier.
III. **LAD Decision Tree;** a classifier for generating a multi-class alternating decision tree using the LogitBoost strategy.
IV. **Support Vector Machine - SVM;** a SVM based classifier with radial basis function kernel.
V. **Neural Network - MLP;** a classifier that uses back-propagation to classify instances.

We used several different classifiers, and found that with a 2-Class K-nearest neighbors IBK classifier we obtain the best detection results with $FP_{rate} = 0.6\%$ and $TP_{rate} = 91.9\%$. This is not an unusual phenomenon in machine learning, that a simple classifier like the IBK performs significantly better than more sophisticated and complex classification methods like neural networks (18; 40). The Receiver Operating

Table 2. Model Selection for Anax 2-Class Classifier in two modes; standalone and "in-line" with the CIDR analysis module

Families	CIDR and Classifier $TP\%$ / $FP\%$ / $Preci.$	Classifier only $TP\%$ / $FP\%$ / $Preci.$
NBayes (Poi)	94.1% / 63.4% / 15.1%	95.0% / 28.9% / 55.4%
IBK (Poi)	**91.9% / 0.6% / 94.6%**	96.4% / 2.7% / 93.1%
SVM (Poi)	57.0% / 0.9% / 88.6%	81.9% / 5.9% / 83.9%
MLP (Poi)	34.4% / 0.8% / 83.8%	54.2% / 3.7% / 84.8%
LAD (Poi)	73.9% / 3.6% / 70.8%	81.5% / 7.4% / 80.7%

Characteristic (ROC) curves for the poison class while using the IBK classifier can be seen in Figure 8.

The reader should note that the $FP_{rate} = 0.6\%$ and $TP_{rate} = 91.9\%$ are **not** packet rates. ROC analysis usually works on rates of detection over network traces, but doing so would unfairly bias the classification results in Anax's favor because the vast majority of the packets are benign. By the definition of the false positive rate (incorrectly classified negatives over total negatives), the number of negatives (or benign packets) is significantly higher than the very sporadic cases of poisoning. Therefore, we decided to instead conservatively calculate the FP_{rate} and the ROC curve based on the unique RRs. In this case, the 0.6% of false positive rate means that for every 1000 unique benign RRs Anax observes for a zone, the poisoning detection system will misclassify six of them as poisonous. To further place the FP_{rate} results into real world context we can look into the domain name "ebay.com", where Anax classified 137 unique RRs over the period of eight months, which means that over an eight month period of time it would misclassify less than a single RR. This indicates that Anax is able to produce low false positive rates due not to the relative volume of the negatives, but due to the accuracy of the 2-Class classifier.

The goal of the 2-Class classifier is to lower the FP_{rate} inherent to the CIDR analysis module due to the limitations of white and black lists. At the same time, we need to

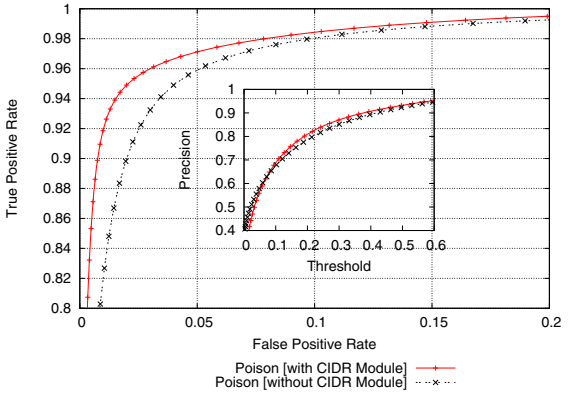


Fig. 8. The ROC curve for poisoning detection in Anax

keep TP_{rate} as high as possible. We observe that when the modules are "in-line", both the FP_{rate} and TP_{rate} are typically better. The only exception is the case of the Naive Bayes (NBayes) classifier where the TP_{rate} decreases in the "in-line" mode. Unfortunately, NBayes cannot be considered as a candidate for our modeling due to the very high FP_{rate} that exhibits in both modes. The "in-line" mode is typically better since the majority of RRs escaping the CIDR analysis module will have the following two characteristics.

First, they are not commonly seen in RRs for the monitored zones. Our whitelist will have very small visibility of the whitelisted address space because we do not risk re-probing and re-verifying correct answers from a small set of trusted recursive servers. In general, maintaining a whitelist has proven to be a very inefficient task. Instead, we use the classifier to leverage the task of identifying other whitelisted RRs. This is possible because the classification features we used to compute the statistical vectors from the passive DNS database will place these uncommon legitimate RRs closer to legitimate trained vectors due to the history of the given IP (present in the newly observed RR) within the passive DNS database.

The second category of RRs that will escape the CIDR analysis module will inevitably contain IPs that belong to CDNs and mainly serve news sites. CDNs tend to fluctuate the network addresses that they use to ensure better quality of service to the end-user of the domain. Static whitelisting cannot keep up with these frequently changing addresses so the CIDR analysis module will not be able to whitelist all CDN addresses. Anax successfully addresses this issue in the 2-Class classification module. IPs from CDN networks produce vectors that are very distinct. Such IPs tend to be mapped to a large number of distinct domain names historically. This list of domain names also shows very small diversity in the number of unique 2LDs and large participation of typical domain names (2LDs) directly correlated with CDNs (e.g., akamai.net, cloudfront.net, llnwd.net). A portion of some CDN related vectors will always be present in the training dataset and the classifier will have no problem correctly classifying similar statistical patterns in the testing dataset.

Anax utilizes passive DNS data for computing its statistical features, therefore it is sensitive to the relative passive DNS window (how long are retained passive DNS data) and how the passive DNS data are aggregated. Operators should collect passive DNS data below the resolver in order to protect their database against out-of-Bailiwick RRs. Furthermore, the utilization of past-CDN IP address space for poisoning could be a significant evasion threat for Anax if the passive DNS window is more than a few weeks. If the window is on the order of several months, then any past-CDN IP address space will still contain past-CDN signal, (considered benign by Anax). This increases the difficulty in identifying poisoning attempts with IPs originating from such addresses.

6 Conclusion

Recently discovered flaws in the DNS protocol require new, innovative techniques to detect poisoning. We have suggested and explored a new area for such research: the detection of DNS poisoning using network observations. We built a system, Anax, that

aims to examine the nature of cache poisoning attacks. Anax is able to detect cache poisoning locally and in a fully automated manner.

Leveraging the fact that DNS poisoning is an inherently localized attack, Anax provides useful insights into attacks, based largely on limited whitelisting and statistical IP and domain name metrics. Anax's detection engine shows that these heuristics can be refined, and placed in order to yield a low (RR-based) FP_{rate} (0.6%), high (RR-based) TP_{rate} (91.9%). Our work has focused on "zones of interest" that are historically targets of phishing attacks.

Anax relies on a fundamental observation about DNS: benign DNS records from major zones generally direct users to a known, usually stable set of `NS-type` and `A-type` records. Poisonings on the other hand generally point victims to new IP addresses. Anax utilizes detection heuristics based on historic passive DNS observations and is able to accurately model benign and malicious RRs. The eight month, real world evaluation shows that Anax is an effective and efficient real-time poisoning detection system.

Acknowledgments

We thank Robert Edmonds and Paul Royal for their valuable comments. This material is based upon work supported in part by the National Science Foundation under grant no. 0831300, the Department of Homeland Security under contract no. FA8750-08-2-0141, the Office of Naval Research under grants no. N000140710907 and no. N000140911042. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Department of Homeland Security, or the Office of Naval Research.

References

[1] Nessus: The network vulnerability scanner, `http://www.nessus.org/nessus/`
[2] OzymanDNS: Kaminsky DNS tunnel (2005), `http://www.doxpara.com`
[3] DNS multi vendor patch: CVE-2008-1447 (March 2008), `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1447`
[4] CERT Advisory. Vulnerability Note VU-23495 - DNS implementations vulnerable to denial-of-service attacks via malformed DNS queries (August 2001)
[5] Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., Feamster, N.: Building a Dynamic Reputation System for DNS. In: Proceedings of the 19th USENIX Security Symposium (August 2010)
[6] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: RFC 4033 - DNS Security Introduction and Requirements
[7] Arends, R., Austein, R., Larson, M., Massey, D., Rose, S.: RFC 4034 - Resource Records for the DNS Security Extensions (2005), `http://www.ietf.org/rfc/rfc4034.txt`
[8] Bellis, R., Phifer, L.: Test report: DNSSEC impact on broadband routers and firewalls (2008), `http://download.nominet.org.uk/dnssec-cpe/DNSSEC-CPE-Report.pdf`
[9] Bernstein, D.J.: Introduction to DNSCurve (2008), `http://dnscurve.org/`

[10] Ccais/RNP (Brazilian Research Network CSIRT) and Vagner Sacramento. Vulnerability in the sending requests control of Bind versions 4 and 8 allows DNS spoofing (November 2002)
[11] Callaway, D.: PorkBind - Recursive multi-threaded nameserver security scanner (2008), http://innu.org/~super/#tools
[12] Computer Academic Underground. bailiwicked_domain.rb (2008), http://www.caughq.org/exploits/CAU-EX-2008-0003.txt
[13] Team Cymru. The Darknet Project (2004), http://www.team-cymru.org/Services/darknets.html
[14] Dagon, D., Antonakakis, M., Day, K., Luo, X., Lee, C., Lee, W.: Recursive DNS Architectures and Vulnerability Implications. In: Proceedings of the 16th NDSS, San Diego, CA (2009)
[15] Dagon, D., Antonakakis, M., Vixie, P., Jinmei, T., Lee, W.: Increased DNS Forgery Resistance Through 0x20-Bit Encoding. In: Proceedings of the 15th ACM CCS, Alexandria, VA (2008)
[16] Dagon, D., Provos, N., Lee, C., Lee, W.: Corrupted DNS Resolution Paths: The Rise of a Malicious Resolution Authority. In: Proceedings of 15th NDSS, San Diego, CA (2008)
[17] DNSstufff. DNS Network Tools: Network Monitoring and DNS Monitoring (2008), http://www.dnsstuff.com/
[18] Duda, R., Hart, P., Stork, D.: Pattern Classification, 2nd edn. Wiley-Interscience, Hoboken (2000)
[19] Elz, R., Bush, R.: (July 1997), http://www.faqs.org/rfcs/rfc2181.html
[20] The Measurement Factory. DNS Survey: Cache Poisoners (2008), http://dns.measurement-factory.com/surveys/poisoners.html
[21] Gummadi, K., Saroiu, S., Gribble, S.: King: Estimating latency between arbitrary internet end hosts. In: Procceding of the 2nd ACM SIGCOMM IMW (2002)
[22] ISC. SIE@ISC, http://sie.isc.org
[23] Kaminsky, D.: Black ops 2008: It's the end of the cache as we know it or: "64k should be good enough for anyone" (2008), http://www.doxpara.com/DMK_BO2K8.ppt
[24] Karmasphere. The open reputation network (2006), https://dnsparse.insec.auckland.ac.nz/dns
[25] Klein, A.: BIND 9 DNS Cache Poisoning (2008), http://www.trusteer.com/files/BIND_9_DNS_Cache_Poisoning.pdf
[26] Osterweil, E., Massey, D., Zhang, L.: Observations from DNSSEC deployment. In: Proceedings of the 3rd NPSec (2007)
[27] Perdisci, R., Antonakakis, M., Luo, X., Lee, W.: WSEC DNS: Protecting Recursive DNS Resolvers from Poisoning Attacks. In: Proceedings of DSN-DCCS, Estoril, Lispon, July 2 (2009)
[28] The Spamhaus Project. Lasso: The Spamhaus Don't Route Or Peer List (2008), http://www.spamhaus.org/drop/drop.lasso
[29] The Spamhaus Project. PBL: The Policy Block List (2008), http://www.spamhaus.org/pbl
[30] The Spamhaus Project. XBL: Exploits block list (2008), http://www.spamhaus.org/xbl
[31] WIDE Project. The TOTD ('trick or treat daemon') dns proxy (January 2006), http://www.vermicelli.pasta.cs.uit.no
[32] Samosseiko, D.: The PARTNERKA - What is it, and why should you care? In: Proceedings of USENIX, Workshop on Hot Topics in Cloud Computing (2009)
[33] Schuba, C.: Addressing weaknesses in the domain name system protocol. Master's thesis, Purdue University (1993)

[34] Ulevitch, D.: Phishtank: Out of the Net into the Tank (2009), http://www.phishtank.com/
[35] USDJ. Eugene E. Kashpureff pleaded guilty to unleashing malicious software on the internet (July 1997)
[36] Vixie, P.: RFC 2671 - Extension Mechanisms for DNS, EDNS0 (1999), http://www.faqs.org/rfcs/rfc2671.html
[37] Vixie, P.: DNS complexity. ACM Queue 5(3) (April 2007)
[38] Wendlandt, D., Andersen, D., Perrig, A.: Perspectives: Improving ssh-style host authentication with multi-path probing. In: Proceedings of the Usenix ATC (June 2008)
[39] Wessels, D.: DNS Cache Poisoners Lazy, Stupid, or Evil? (2002), http://www.nanog.org/mtg-0602/pdf/wessels.pdf
[40] Witten, I., Frank, E.: Data mining: practical machine learning tools and techniques. In: Morgan Kaufmann Series in Data Management Systems. Morgan Kaufman, San Francisco (June 2005)
[41] Yuan, L., Kant, K., Mohapatra, P., Chuah, C.: DoX: A Peer-to-Peer Antidote for DNS Cache Poisoning Attacks. In: ICC 2006 (2006)

Behavior-Based Worm Detectors Compared*

Shad Stafford and Jun Li

University of Oregon
{staffors,lijun}@cs.uoregon.edu

Abstract. Many worm detectors have been proposed and are being deployed, but the literature does not clearly indicate which one is the best. New worms such as IKEE.B (also known as the iPhone worm) continue to present new challenges to worm detection, further raising the question of how effective our worm defenses are. In this paper, we identify six behavior-based worm detection algorithms as being potentially capable of detecting worms such as IKEE.B, and then measure their performance across a variety of environments and worm scanning behaviors, using common parameters and metrics. We show that the underlying network trace used to evaluate worm detectors significantly impacts their measured performance. An environment containing substantial gaming and file sharing traffic can cause the detectors to perform poorly. No single detector stands out as suitable for all situations. For instance, connection failure monitoring is the most effective algorithm in many environments, but it fails badly at detecting topologically aware worms.

Keywords: Internet worm, worm detector, behavior-based detection.

1 Introduction

Network worms have long posed a threat to the functioning of the Internet. As early as the outbreak of the Morris worm in 1988 [1], they have been capable of disrupting traffic over large swathes of the Internet. Significant outbreaks such as the CodeRed [2] and Slammer [3] worms in 2001 and 2003 brought the threat to national prominence and spurred the development of a wide range of mechanisms to detect the presence of worms and to harden operating systems against common attacks. The emergence of the Conficker worm [4] in late 2008 showed that those efforts had not eradicated worms completely. As the Internet continues to play a more important role in everyday life for hundreds of millions of people and as the very nature of the devices on the Internet is changing (e.g., consumer-level mobile devices begin to make up a substantial portion of connected devices), the Internet requires more protection than ever. The question remains—can we protect our networks from worms?

* This material is based upon work supported by the United States National Science Foundation under Grant No. CNS-0644434. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 38–57, 2010.

The relevance of this question was further highlighted in late 2009 when a network worm was found propagating exclusively on iPhones. The IKEE.B worm [5] takes advantage of a default root password set in some *jail-broken* iPhones to propagate. It brings to light some shortcomings in current worm detection and prevention work. Specifically, because it propagates via an encrypted channel, it bypasses worm detectors that rely on examining the content of network traffic; and because it exploits a configuration error rather than a buffer overflow to gain control of the target machine, it is undeterred by defensive techniques such as address space randomization.

In this paper, we examine our ability to detect the presence of a worm in a protected network. Existing detection schemes can be broadly classified into host-based systems that monitor system call information or other host-level behavior for illegal operations, and network-based systems that monitor network traffic. Network-based systems can be further broadly divided into content-based systems that monitor the bytes transmitted across the network and behavior-based systems that monitor the patterns of network traffic.

Unfortunately, it is unclear how these detection systems perform relative to each other as there is very little work that directly compares them. Algorithms are typically published with evaluations against a single network trace, which is different for different algorithms and generally not available publicly. For example, the MRW detector was evaluated against an unidentified week-long trace of a university department with 1,133 identified hosts [6], whereas the TRW algorithm was evaluated against two traces collected at the peering link of an ISP containing 404 and 451 identified hosts [7]. Furthermore, worm detectors are evaluated with different performance metrics, and tested worms do not always follow the same set of parameters (such as scanning strategy and speed). For example, the detection latency of the DSC detector is measured in the percent of the network infected at detection time [8] while the MRW detector does not provide detection latency results at all.

We seek to remedy this situation by performing a comprehensive analysis of several worm detectors that are easily deployable and in principle capable of detecting IKEE.B. We select six of the most prominent behavior-based worm detection techniques and measure their detection performance against a variety of worm propagation strategies over a common set of network traces. We evaluate each detector using key performance metrics related to accuracy and latency. The questions we seek to answer include: Is any one detection algorithm clearly superior to the others, including cases when fast worms are the only concern or a special network environment is protected (e.g., residential networks that see game or peer-to-peer network usage)? If a worm adopts smart scanning strategies such as slowing down or intelligently choosing victims, can it evade these detectors? And, does the network trace selected for evaluation significantly impact the detection performance?

Highlights of our findings include: (1) We find that the network trace impacts the sensitivity of the detectors. They are less sensitive in environments with more Internet gaming and file sharing activity, which appears more similar to worm

activity than other benign activities such as web browsing. (2) Our results show that there is no clear winner and every detector has its limitations. For example, connection-failure monitoring is the most consistently sensitive detection technique for random scanning and local-preference worms, but it fails drastically in the case of a topologically aware worm. (3) In all environments, a stealthy worm scanning at one scan per minute and employing some form of topologically aware scanning that avoids connection failures could evade all the detectors evaluated in all environments.

The rest of this paper is organized as follows: We first discuss how we selected detectors in Section 2, and then examine the selected detectors in some detail in Section 3. We discuss our selected metrics in Section 4, followed by the methodology by which we evaluate detectors in Section 5. We present the results of our evaluations in Section 6. Section 7 reviews related work, with our conclusions in Section 8.

2 Detector Selection

In this section we describe published worm detection algorithms and justify our choice of six specific detectors for this comparison study. We performed an extensive evaluation of proposed worm detectors, considering 36 different published works. We grouped them into the following categories based on their detection algorithm: host-based detectors, content-based detectors, and behavior-based detectors. Each category has its own strengths and weaknesses.

Detectors that we classified as *host-based* included, among others: COVERS [9], DACODA [10], TaintCheck [11], and Sweeper [12]. Several factors, however, lead us to exclude host-based detectors from this study. Host-based detectors require end-host deployment but a network operator may have no control over what software is installed on the end-hosts running in their network. Furthermore, users may circumvent host-based software installs as illustrated by IKEE.B, which targeted only those users who intentionally installed an unsupported operating system. Finally, it is unclear whether host-based systems are capable of detecting an attack like that used by IKEE.B. The systems listed above all rely on observing malicious memory manipulations such as buffer overflows, but IKEE.B did not perform any illegal memory operations; it merely exploited a configuration vulnerability.

Detectors that monitor the network instead of end-hosts seem much more promising because they do not require deployment on each host to be protected. We first look at detectors that examine the contents of network traffic, including AutoGraph [13], EarlyBird [14], PAYL [15], Anagram [16], and LESG [17]. Each of these detection mechanisms share a similar limitation that leads us to exclude them from our comparison: *they are unable to monitor encrypted traffic.* Encrypted traffic is a special case of making a worm polymorphic. Content-based systems designed to catch polymorphic worms (such as Polygraph [18]) depend on attack-specific, invariant sections of content which may not be present for an encrypted worm. Even when worms are transmitted using unencrypted connections, advances in polymorphism research such as [19] have threatened the

promise of these detectors. Also, it is prohibitively difficult to acquire a variety of network traces which contain full network content, making it infeasible to evaluate these detectors.

The remaining and largest class of detectors is behavior-based (or payload oblivious) detectors. These include TRW [7], RBS [20], PGD [21], and many others. These systems also monitor network traffic, but they examine the behavior of traffic from end hosts rather than the contents of their packets. This type of system is easily deployed, requiring as little as a single monitor at the network gateway. They are capable of detecting worms regardless of the scanning mechanism or propagation type (including propagation via encrypted channels), and many of them are capable of identifying the worm-infected hosts. However, we do exclude some behavior-based systems that a network operator could not easily deploy. For example, detectors using network telescopes (such as those by Wu et al. [22] and Zou et al. [23]) require a large dark address space and cannot be deployed by a network operator unless they control a large address space.

After our exhaustive evaluation of worm detectors, we are left with the following selections: TRW [7], RBS [20], TRWRBS [20], PGD [21], DSC [8], and MRW [6]. We discuss these detectors in greater detail in the next section.

3 The Selected Worm Detectors

Having selected detectors for our comparison work, we now describe them each in more detail in roughly chronological order of their publication. We present only a brief a summary of each work, please refer to the original publications for more detail. Note we used existing acronyms for each work where available.

The TRW detector was published by Schechter et al. in 2004 [7]. TRW identifies a host as worm infected if connection attempts to new destinations result in many connection failures. TRW is based on the idea that a worm-infected host that is scanning the network randomly will have a higher connection failure rate than a host engaged in legitimate operations. Even with the IPv4 address space getting closer to complete allocation, the majority of addresses will not respond to a connection attempt on any given port. Randomly targeted connections (as in worm scanning) will likely fail.

The destination-source correlation detector (DSC) was published in 2004 by Gu et al. [8]. It detects a worm infection by correlating an incoming connection on a given port with subsequent outgoing infections on that port. If the outgoing connection rate exceeds a threshold established during training, the alarm is raised. A different threshold is maintained for each destination port.

The MRW detector was first published in 2006 [6]. It is based on the observation that whereas worm scanning results in connections to many destinations, during legitimate operations the growth curve of the number of distinct destinations over time is concave. And as the time window increases, destination growth slows. This can be leveraged by monitoring over multiple time windows with different thresholds for each window. If the number of new destinations for a host within a given window exceeds the threshold, the alarm is raised.

The RBS detector was first published in 2007 [20] by Jung et al. . Similar to the MRW detector, RBS measures the rate of connections to new destinations. The work is based on the hypothesis that a worm-infected host contacts new destinations at a higher rate than a legitimate host does. RBS measures this rate by fitting the inter-arrival time of new destinations to a exponential distribution.

The TRWRBS detector was published alongside the RBS detector [20]. It combines the TRW and RBS detectors into a unified scheme, and observes both the connection failure rate and the first contact rate. It performs a sequential hypothesis testing on the combined likelihood ratio to detect worms.

The Protocol Graph detector (PGD) was introduced by Collins and Reiter in 2007 [21]. It is targeted at detecting slowly propagating hit-list or topologically aware worms. PGD works by building protocol-specific graphs where each node in the graph is a host, and each edge represents a connection between two hosts over a specific protocol. Collins and Reiter made the observation that during legitimate operations over short time periods, the number of hosts in the graphs is normally distributed and the number of nodes in the largest connected component of each graph is also normally distributed. During a worm infection, however, both numbers will go beyond their normal range, indicating the presence of the worm.

4 Performance Metrics

The goal of this study is to evaluate the selected detectors over a comprehensive parameter space to identify their strengths and weaknesses. We must first, however, determine which performance attributes we are most interested in capturing, and what metrics would be suitable for assessing them.

The focus of this study is on the ability of the detectors to discover the presence of a worm in the network. We thus want to measure their accuracy: does a detector alert us when a worm is present—but not do so when there is no worm? Furthermore, we want to measure its ability to detect a broad range of worm scanning algorithms. Moreover, accurate detection is not helpful if it happens too far after the fact. We must obtain some notion of the speed of the detectors—does it find a worm quickly or does it allow the worm free action for a long time before raising the alarm.

There are some attributes that we are not as interested in. At this time we are ignoring runtime costs such as processing or memory requirements. These are dependent on implementation and optimization details, and can vary widely for a given detection algorithm (for example, see the hardware implementation of TRW by Weaver et al. [24]). It is beyond the scope of this work to attempt to determine how efficiently each of these algorithms could be implemented. Similarly, we do not consider the complexity of installing or running the detector. This is not because installation complexity does not impact the potential adoption rate of a detector, but because it is somewhat orthogonal to the accuracy of the detector itself and could be addressed separately from the detection algorithm itself.

As shown in Table 1, we have identified four metrics as the most useful measures of the performance of a worm detector. We explain them below:

Table 1. Metrics

F-	Percentage of experiments where worm traffic is present but not detected in time period τ
F+ by host	The number of false alarms raised during a time period τ, limited to at most one false alarm per host
F+ by time	Percentage of minutes during a time period τ where a false alarm is triggered for any host
Detection Latency	The number of outbound worm connections from an infected network prior to detecting the worm

Our false negative metric works as follows. For each experiment we introduce a worm to the background legitimate traffic. The detector is limited to a time period τ (typically an hour) to detect the worm after it becomes active. If in that time span an alarm is not raised, the experiment is scored as a false negative for the detector. The *false negative rate* (F-) is the percentage of experiments scored as false negatives. (We report F- for each different scanning rate of the worm.)

The flip side of false negatives is false positives: reporting legitimate traffic as a worm infection. This is a critical metric for worm detectors, because a detector that repeatedly raises a false alarm ("cries wolf") will quickly be ignored by network administrators. We measure false positives by running the detector against benign traffic with no injected worm activity. (Because we have inspected the traces for known worm activity, we consider every alarm raised by a worm detector a false alarm.) However, because worm detectors often repeat their worm infection tests—on every connection in some cases, the same set suspicious behavior may cause the alarm to be raised repeatedly, and these repetitive alarms should be coalesced into a single notification to the network administrators. But the exact mechanism and scope of alarm coalescing will be specific to the needs and resources of the network administrators at each site. As a result, we present two forms of false positive rate. We present the number of hosts identified as infected (coalescing alarms by network address) as the *false positive rate by host* (F+ by host). We also define *false positive rate by time* (F+ by time), which is the fraction of minutes of the trace where an alarm is raised on at least one host; note the alarm duration is only until the end of the current minute as we coalesce alarms into 1 minute bins. The combination of these two metrics give a better view of the overall false positive performance of the detector than either does individually.

The next major performance attribute to consider is the speed with which a worm is detected. The faster detection occurs, the less damage the worm can do. We measure *detection latency* as the number of outbound worm connections

initiated by all infected hosts in the protected network prior to detection of any internal infection. (Scans that do not leave the network do not inflict damage on the Internet as a whole and are not included in this count.) Alternative approaches such as using clock time or infected host count are less accurate and less descriptive than our metric.

5 Experiment Design

We run the detectors against legitimate traffic to measure false positives, then against legitimate traffic plus known worm traffic to measure false negatives and detection latency. We developed a custom testing framework and implemented each detector in our framework based on the detector's published specifications. Our framework can run against online, real-time traffic on the DETER testbed [25], as well as run in an offline (not real-time) mode. We use legitimate traffic from a variety of sources and generate known worm traffic by simulating a worm with our GLOWS [26] simulator. We vary the following parameters as we evaluate each worm detector: the environment it is run in (meaning the network configuration and legitimate traffic), the worm scanning method, and the worm scanning rate. We have also studied the effects of two additional parameters: the target port attacked by the worm and the activity profile of the first host infected by the worm, but omit those results from this paper due to space constraints.

5.1 Evaluation Environment and Background Traffic

Worm detectors must be evaluated in the context of a subnet to be protected and against the legitimate background traffic that occurs in that subnet. For our experiments, we define an *environment* as the network address space to be monitored, the IP addresses of the active hosts inside that address space, and the IP network traffic into and out of that address space during two time periods. We use the first time period for training and the second to run experiments against. To make the environments comparable to each other and to enable us to ensure that they do not contain worm traffic, we select a /22 subnet from the original recorded traces to use as the protected address space in our environment. Every environment is thus a /22 network with between 100 and 200 active hosts. We use four distinct environments in our evaluation.

The *enterprise* environment is built from a trace collected at LBNL [27] in January of 2005. Heavy scanners were removed from the trace before it was released. It has 139 active hosts and the training and experiment segments each contain roughly 25,000 connections.

The *campus* environment is built from a trace that was collected in 2001 at the border of Auckland University [28]. The trace was anonymized using a non-prefix preserving anonymization scheme, so we cannot entirely accurately reconstruct the internal structure of their network. Instead, we randomly select 200 hosts and construct an environment using traffic to and from those hosts. Each segment of the trace in our campus environment contains approximately 25,000 connections.

The *wireless* and *department* environments are built from traces collected at the University of Massachusetts in 2006 [29]. The department environment is built from a trace capturing all traffic to and from the wired computers in the CS department. It has 92 active hosts and approximately 30,000 connections in each segment. The wireless environment comes from a trace capturing all wireless network traffic from the university. It has 313 active hosts and approximately 120,000 connections in each segment.

5.2 Worm Parameters

Several key parameters of a worm may impact the effectiveness of worm detectors. We look at three scanning strategies worms can employ: random scan, local-preference scan, and topologically aware (topo) scan, and evaluate them at a variety of scanning rates. Our GLOWS simulator takes an environment as input and simulates a worm as if it were attacking the network defined by that environment. The simulation starts with a single inbound worm connection that infects one host in the protected network. We run the simulator once for each permutation of worm parameters. The scanning mechanisms are defined as follows.

A random scanning worm simply chooses target addresses at random from the entire IPv4 address space. This typically results in many connection attempts to addresses with no host present or with a host that is not running the requested service, resulting in many connection failures. Permutation and sequential scanning worms should show very similar characteristics and are not evaluated separately here.

A local-preference worm scans local addresses (in the same prefix) more frequently than addresses in the full address space. This results in more scans that do not cross the network border (and are therefore not visible to a border-located detection mechanism). Existing local-preference scanning worms, such as Code-Red II [2], target the local /16 prefix approximately 50% of the time, the local /8 25% of the time, and the entire network the remaining time. As all our traces are about a /22 network, such a worm would largely resemble a random scanning worm. Instead, our local-preference worm scans the local /22 50% of the time, the local /8 25% of the time, and the entire network the remaining time.

The topologically aware (topo) worm finds target information on the host that it infects. This target information allows it to scan effectively because it already knows about other hosts that are running the service it targets. The number of new hosts (referred to as "neighbors") the worm discovers is dependent on its neighbor detection algorithm. We use three implementations of the topo worm with differing neighbor counts. The topo100 worm starts with 100 neighbors, the topo1000 worm starts with 1000 neighbors, and the topoall worm starts with an unlimited supply of neighbors. After scanning its known neighbors, the topo worm must either stop scanning or switch algorithms. In our implementation it reverts to random scanning after exhausting its neighbor list. Note that the neighbors discovered by the topo worm are randomly located, so could appear both inside and outside the protected network. Also, they will be running the target service but are not guaranteed to be vulnerable.

In addition to scanning mechanism the worm uses, the rate at which it initiates connections is important. The faster a worm scans, the more visible it is to worm detectors. We run experiments for a variety of worm scanning rates ranging from 10 connections per second down to one connection every 200 seconds.

5.3 Experiment Procedure

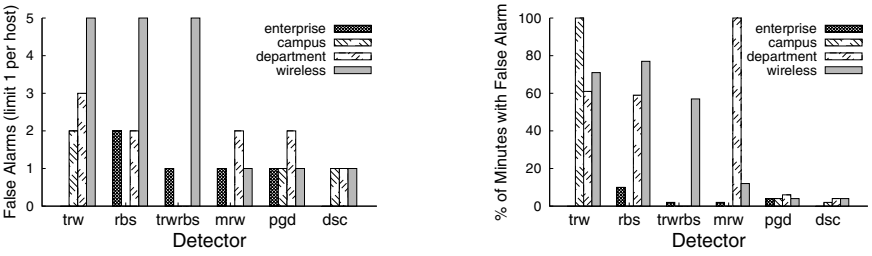
Measuring detector performance is a multi-step procedure. For each environment, every detector must (1) establish thresholds via training, (2) be evaluated against the legitimate traffic in the environment to measure false positives, (3) adjust their parameters to fix false positives at a specific level, and (4) be evaluated against legitimate traffic combined with worm traffic to measure false negatives and detection latency. Let us now discuss each of these steps in more detail.

These detectors are anomaly detectors, and they look for traffic that diverges from normal. To do this, they must first measure what normal is. The TRW, MRW, DSC, and PGD detectors are run against the training segment of the trace using the training method outlined in their publication to perform this operation. The RBS and TRWRBS detectors perform on-the-fly training as they are run against the experiment segment of the trace.

After the thresholds are established from the training segment of the trace, each detector is run against the experiment portion of the trace to measure false positives. We measure F+ using the thresholds obtained from training and the default detector parameters outlined in the original publication of each work, presenting those results in Section 6.1.

Note that each detector can be tuned to favor producing either more F+ or more F-. After reporting F+ using the default detector parameters as published, in order to provide a fair comparison of the false negative rate of the detectors, we modify each detector's parameters such that they all produce the same number of false positives in each environment. We chose to peg each detector at a rate of two false positive alarms during the experiment period. Two false positives is a high rate for the one-hour time period evaluated, but was chosen as an achievable value for all detectors requiring the minimum amount of parameter modifications.

After measuring F+ and adjusting the detectors to match their F+ levels, we then measure the performance of the detectors against worm traffic. For each detector in each environment, we run 16 experiments for every permutation of the worm parameters. A single experiment consists of running the detector for 10 minutes of the experiment trace to warm up the connection histories, then injecting the simulated worm traffic into the trace, and running until either an hour has elapsed or the worm is detected. Each of the 16 experiments that we run for a given set of worm parameters has a different host in the protected network being infected first and uses a different random seed. The percentage of experiments where the worm is not detected is the false negative rate, and the mean number of worm connections that have left the network at detection time is the detection latency.



(a) By Host (b) By Time

Fig. 1. False positives against legitimate traffic: when running with default parameters against the experiment segment of the traces with no worm traffic injected

6 Results

We now measure the performance of the worm detectors in a variety of worm scenarios. We first look at the false positives, then introduce worm traffic to measure false negative rates and detection latency. We start with the simplest worm strategy of randomly scanning addresses, then increase the worm sophistication to local-preference and then topologically aware scanning strategies.

6.1 False Positives against Legitimate Traffic

Figure 1(a) shows the results for each detector using default parameters from its original publication. Raising an alarm for a host could either (a) indicate that the host is considered permanently infected, or (b) indicate that the host is behaving anomalously *now* (for some definition of now). Figure 1(a) shows F+ results using strategy (a) (with PGD limited to one alarm per 1-minute window because it does not identify the infected host). Figure 1(b) shows F+ results using strategy (b) and with an alarm duration of one minute. Strategy (a) is probably more representative of how detectors would be deployed in practice, but it is illustrative to show that without such a limitation, in some environments RBS and TRWRBS would be in an alarm state more than 50% of the time and TRW and MRW would be in an alarm state 100% of the time.

These results also demonstrate the impact that environment has on the detector performance. TRWRBS has five F+ in the wireless environment but none in the campus or department environments. MRW is in an alarm state 100% of the time in the department environment but not at all in the campus environment. An evaluation using only a single environment could produce grossly inaccurate results.

The wireless environment showed the most F+ activity with the default parameter choices. This appears to stem from several hosts playing network games such as Counter-Strike (UDP connections on ports in the 27010-27050 range) and NeverWinter Nights (TCP connections on port 5121) as well as from hosts using BitTorrent (33 hosts active on ports in the 6881-6999 range). This environment represents the most residential/recreational usage patterns and indicates

that this sort of traffic is less amenable to behavior-based worm detection than the less variable traffic of the enterprise environment. This represents the first findings we are aware of that validate a common hypothesis: current behavior-based anomaly detectors are not optimized for residential style network traffic and may not show satisfactory performance in such an environment.

6.2 Detector Performance against Random Worm

In this section we report false negative and latency results against random scanning worms. Figure 2 shows that TRW is the most consistently effective detector across the environments, discovering all instances of the worm down to 0.05 scans per second and catching the majority of the slower scans in the enterprise and campus environments. RBS is the least effective, only able to consistently detect the worm scan rates greater than five scans per second. TRWRBS blends the two detectors with results right in the middle. The DSC and PGD detectors are an order of magnitude more effective in the enterprise environment than in the other environments due to the lower activity levels (and hence lower thresholds) in the enterprise environment. The MRW detector provides middle of the road performance except against in the wireless environment where it is unable to detect the worm at speeds slower than five scans per second.

Figure 3 shows the average number of connections each infected network was able to make before detection. Note that the scale is not consistent across the graphs. We only show the value for those scenarios where F- is zero in order to eliminate selection bias in the results. DSC is consistently the fastest detection

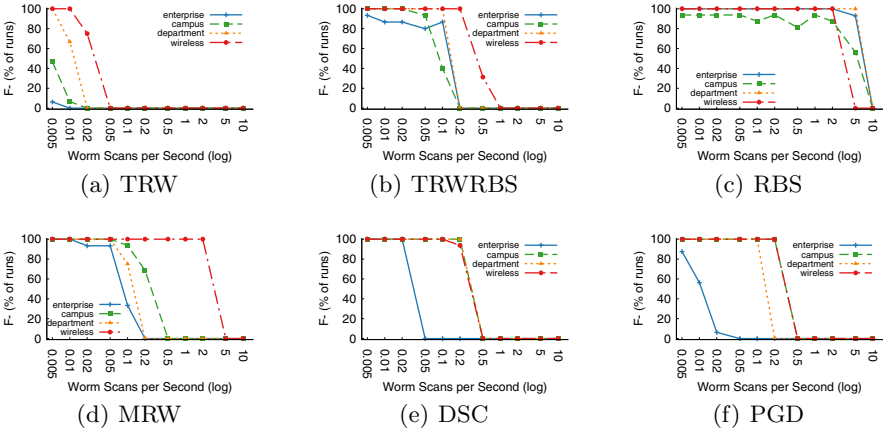


Fig. 2. F- against random worm: percent of experiments where the worm was **not** detected (lower is better performance) with a *random* scanning worm infecting randomly selected hosts. For each environment and scanning rate we conducted 16 individual experiments using different first infected hosts and different random seeds. In each case the experiment was run until the worm was detected or one hour elapsed without detection.

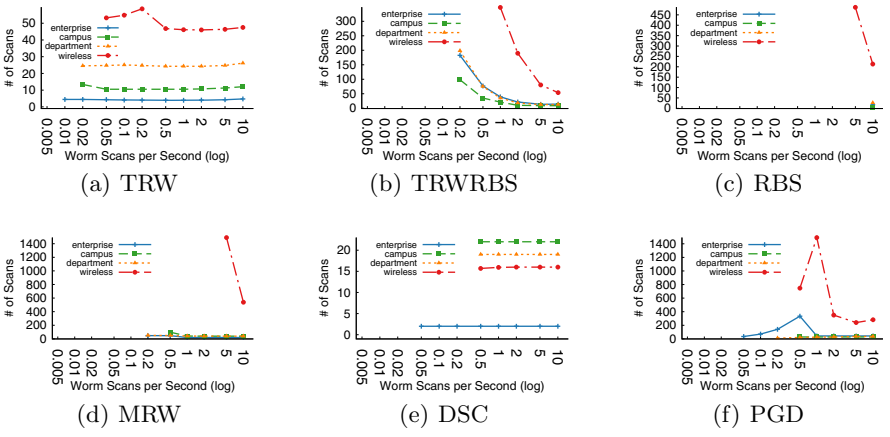


Fig. 3. Latency against random worm: from worm infection time to detection time for *random* scanning worm, measured as the number of worm connections leaving the protected network prior to detection. We report results only for those environments and scan rates where the worm was detected with 100% accuracy.

mechanism, never allowing the worm to scan more than 23 times before detection. TRW again highlights the variation between environments, allowing roughly 50 worm scans in the wireless environment before detection, but only five scans in the enterprise environment. MRW and RBS allowed several hundred scans before detection in the wireless environment, but were much faster in the other environments. PGD showed the most variation, allowing over 1000 scans before detection in some scenarios in the wireless environment but detecting the worm in 30-40 connections in the other environments. TRWRBS showed increasing latency as the scan rate drops. This is due to the influence of the RBS algorithm that increases the destination threshold as the time window increases. The fast scanning worm is caught in a short window, but the slower scanning worms take a substantially longer time to hit the critical number of destinations.

Across the board, TRW shows the best detection performance against random scanning worms. This indicates that connection failures are a strong and highly identifiable signal. TRW also had consistent and low latencies, limiting the damage a worm could do. Destination pattern based detection such as MRW and RBS typically requires greater numbers of connections for accurate identification. PGD performed adequately, but is designed to detect multiple infected internal hosts which did not happen with the random-scanning worm.

6.3 Detector Performance against Local-Preference Worms

Having examined the baseline case using the random scanning worm, we now investigate performance against a more advanced foe: the local-preference scanning worm. The local-preference worm directs half its connections at the local network, meaning both that it is more likely to infect multiple hosts inside the

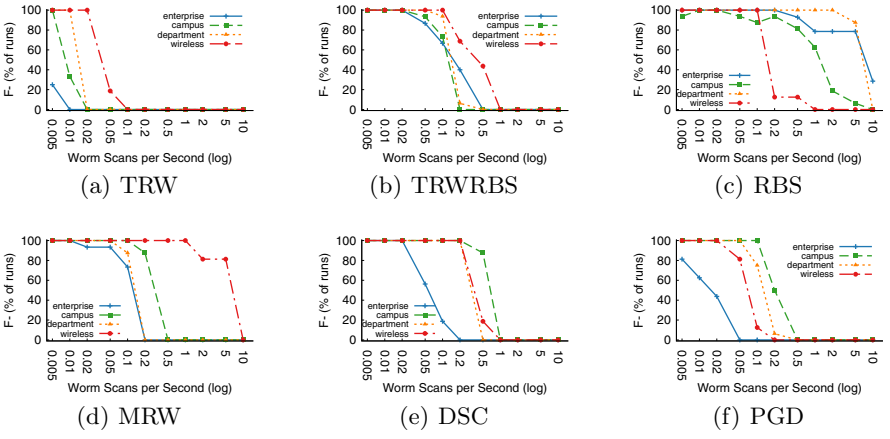


Fig. 4. F- against local-preference worm: percent of experiments where the worm was **not** detected (lower is better performance) with a *local-preference* scanning worm

protected network and that fewer connections per time period are visible to a gateway-based detector. However, the scan is still random in nature, so shares the same general characteristics as the purely random scanning worm.

Figure 4 shows that for most scenarios, the detectors show a slight decrease in sensitivity. This is visible as a shift to the right in the false negative curves. The TRW detector was able to detect 100% of the random worms in the wireless environment at 0.05 scans per second, but is only able to detect 100% of the local-preference worms at 0.1 scans per second. TRWRBS, RBS, MRW, and DSC all show similar decreases in performance in some environments. The reason for this is simply the reduction of worm scans that are visible to the detector. The limit of a detector's ability to spot the worm—meaning the slowest worm that it can detect reliably—is at the point where it can just barely observe enough evidence to infer that a host is infected. If a worm scans more slowly or not all its scans cross the gateway (as in local-preference worms), the evidence visible to the detector may not be enough to make the determination that a worm is present.

The one detector that shows a significantly different response is the PGD detector, showing *better* performance against the local-preference worm than it did against the random worm. The PGD detector measures the protocol graph of all hosts in the network, and the more infected hosts there are, the more scanning there will be using the protocol the worm targets. This leads to either more total nodes in the graph or a larger connected component, allowing the PGD detector to spot the local-preference worm in situations where it would not have detected a random scanning worm.

The latency results are also impacted by the local-preference scanning strategy (Figure 5). The TRWRBS, RBS, DSC, and MRW detectors show worse detection latency in all environments for the local-preference worm as compared to

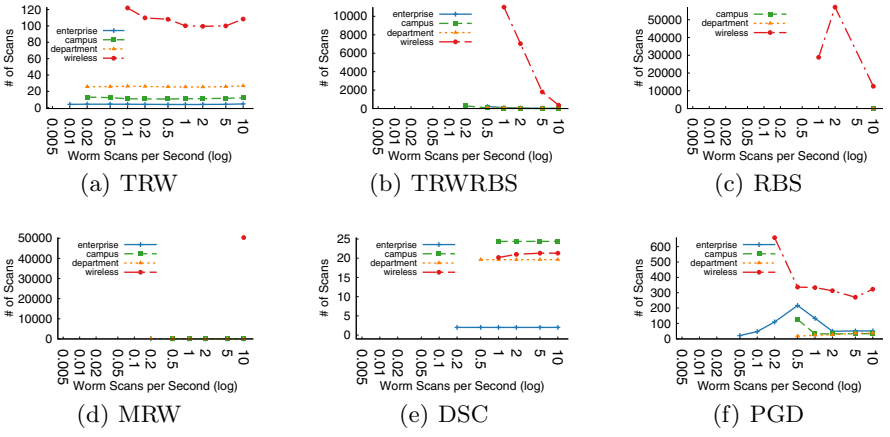


Fig. 5. Latency against local-preference worm: from worm infection time to detection time for *local-preference* scanning worm, measured as the number of worm connections leaving the protected network prior to detection

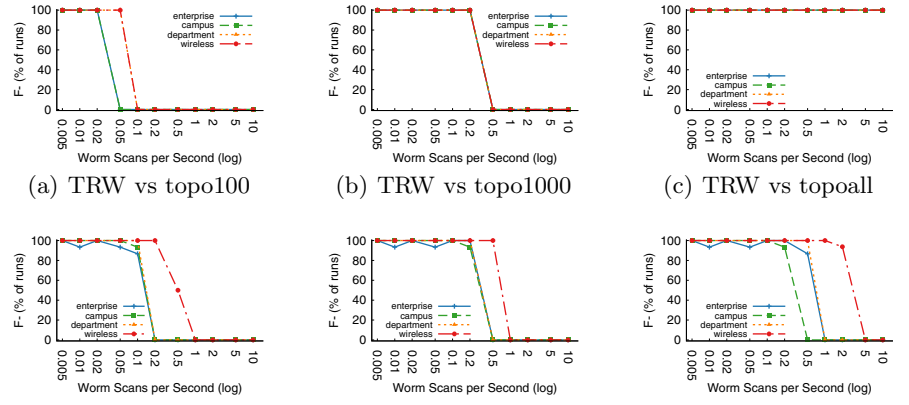
the random worm. This is because the worm targets the local network so aggressively that in many scenarios it infects multiple hosts inside the network before it is detected. Recall that our latency metric measures the combined external scanning of *all* infected hosts in the network. The TRW detector, on the other hand, shows identical latency performance for all environments when comparing random and local-preference worms because it detects the worm before it infects *multiple* hosts (except in the wireless environment).

PGD behaves quite differently than the other detectors. It detects the local-preference worm more quickly than the random worm in the enterprise and campus environments, but slower in the department environment. And in the wireless environment the local-preference worm is detected more quickly at scanning rates of two scans per second or less, but the random worm is detected more quickly at rates above two scans per second.

The DSC detector is the fastest, allowing fewer than 25 outgoing worm connections in all scenarios where it was able to detect the worm 100% of the time. TRW is also quite fast, allowing fewer than 27 connections in all environment except for the wireless environment where it allows roughly 100. Note TRW also is the most sensitive detector, successfully detecting the worm at the lowest scanning rates in all environments.

6.4 Detector Performance against Topo Worms

Topo scanning changes the observed behavior of an infected host by reducing the number of connection failures that the detector can observe. The neighbors discovered by the topo worm are vulnerable at the same level as other hosts in the network but are guaranteed to be present, different from random scanning where



(a) TRW vs topo100 (b) TRW vs topo1000 (c) TRW vs topoall

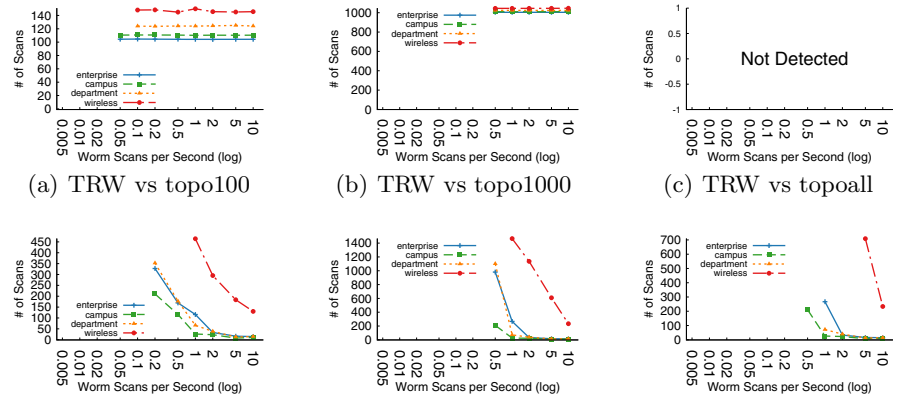
(d) TRWRBS vs topo100 (e) TRWRBS vs topo1000 (f) TRWRBS vs topoall

Fig. 6. F- against topo worm: percent of experiments where the worm was **not** detected (lower is better performance) by the TRW and TRWRBS detectors with a *topo* scanning worm. The topo100 worm uses 100 neighbors before reverting to random scanning, the topo1000 worm uses 1000 neighbors before reverting to random scanning, and the topoall worm never uses random scanning.

a large number of scans go to addresses with no host present. The only detectors that are impacted by this strategy are those detectors that rely on observing connection failures: TRW and TRWRBS. The RBS, MRW, DSC, and PGD detectors show identical performance against the topo worm and the random worm. The pattern of neighbors—whether they can be connected to or not—is random in both the random and topo worms and thus triggers those algorithms in the same way.

The TRW detector is unable to detect the topo worm during its topo scanning phase because of the lack of connection failures. It *only* detects the worm after it reverts to random scanning. In the topo100 scenario (Figure 6(a)), this occurs relatively quickly as it does not take long for the worm to exhaust its list of 100 neighbors. TRW is able to detect the worm at speeds as low as 0.01 scans per second in all environments. However, in the topo1000 scenario, the list of neighbors is not exhausted during the one-hour experiment for speeds below 0.5 scans per second and the TRW detector is unable to detect topo worms with slower scanning rates (Figure 6(a)). In the topoall scenario—where the topo worm never exhausts its list of neighbors—the TRW detector is *never* successful at detecting the worm (Figure 6(c)).

Not only is TRW's ability to detect the worm compromised, but even in scenarios where it does detect the worm it is much slower at it. Figures 7 show the latency results for TRW against the topo worm. Because during the worm's topo phase none of its scans were detected, the latency results against the topo100 worm are approximately 100 scans worse than they were for TRW against the random scanning worm. Similar results can be seen for the topo1000



(a) TRW vs topo100 (b) TRW vs topo1000 (c) TRW vs topoall

(d) TRWRBS vs topo100 (e) TRWRBS vs topo1000 (f) TRWRBS vs topoall

Fig. 7. Latency against topo worm: from worm infection time to detection time for *topo* scanning worm, measured as the number of worm connections leaving the protected network prior to detection. The topo100 worm uses 100 neighbors before reverting to random scanning, the topo1000 worm uses 1000 neighbors before reverting to random scanning, and the topoall worm never uses random scanning.

scenario, where TRW's detection latency is 1000 connections worse than it was for the random scanning worm.

This shortcoming in TRW is one of the motivations for the TRWRBS detector. It uses connection failures in the detection algorithm, but it can also detect a worm even with no connection failures by checking the rate of connections to new destinations. The TRWRBS detector is able to detect the topo100 at rates above 1 scan per second in the wireless environment and above 0.2 scans per second in all other environments (Figure 6(d)). It does not perform quite as well as TRW in this scenario because TRW is able to leverage the connection failures so effectively. In the topo1000 scenario the detectors are effective at approximately the same worm scanning rate (Figure 6(e)); but if one looks at the latency, the TRWRBS detector is able to detect the worm more quickly at most scanning rates (Figure 7(e)). At worm scanning rates of 2 scans per second and higher, TRWRBS can detect the worm in under 30 connections in all the environments except for the wireless environment. This compares well against the TRW algorithm which requires over 1000 scans before detecting the topo1000 worm. The TRWRBS detector even detects the worm in the topoall scenario where the TRW detector could not.

This reliance on connection failures highlights a potential weakness of the TRW algorithm. If a worm can generate a big enough list of hosts running the target service that are likely to exist, it can make enough successful connections to completely evade the TRW algorithm. The detectors based on destination distributions do not have this weakness.

6.5 Summary

We now recap our findings and answer the questions posed in the introduction. We found that no detector was clearly superior to the others in the study. The TRW detector can detect slower random and local-preference scanning worms than any of the other detectors in all the environments we tested. However, it performs poorly against topo worms. In fact, a topo worm with a large supply of neighbors to scan is entirely undetectable by the TRW algorithm. The PGD detector was capable of detecting all types of worms scanning at 0.5 scans per second or faster in all environments, but was relatively slow, frequently allowing several hundred scans prior to detection. The TRWRBS detector was similar to the PGD detector, but showed decreased performance against topo worms. The RBS detector was only capable of detecting fast scanning worms. The MRW detector struggled to detect worms in the wireless environment and was incapable of detecting the local-preference worm in that environment. Finally, the DSC detector performed quite well in many respects, but is otherwise quite limited due to the requirement that an inbound infecting connection be observed in order for the detector to function. An initial infection that came via some other vector (removable media, direct download, etc.) would be undetectable by DSC.

If we narrow our criteria, however, we may be able to identify some detectors as being superior at specific tasks. For example, if we only consider fast scanning worms—those that make 10 scans per second—the TRWRBS detector suddenly stands out as being an excellent choice. It detects fast scanning worms in every environment regardless of scanning type and is the fastest in most scenarios.

The wireless environment was the most difficult for detectors to operate successfully in. In virtually all scenarios, detectors showed the worst sensitivity in the wireless environment, and detection latencies were typically an order of magnitude worse. The traffic in this environment is more focused around entertainment type activities such as network gaming and peer-to-peer file sharing. These activities are prone to resembling worm scanning activity, making it more difficult for the detectors to differentiate between legitimate hosts and worm infected ones. For example, a peer-to-peer network client may receive a list of peers who were recently active and attempt to contact every host on the list. If the peer-to-peer network has a high churn rate and hosts on the peer list have left the network, this activity will result in many connection failures, just as if a worm were scanning for potential targets. Even in the face of this type of activity, however, the detectors were still typically able to detect true worm activity. As in the other environments, the TRW detector was able to detect slower worms than any other detectors. The PGD detector showed the next best performance and had the advantage of also detecting the topo worm in the wireless environment.

Our results indicate that worms scanning at one connection per second or better are relatively easily detected in most environments, but a worm that utilizes some sort of topo scanning with a low connection failure rate could evade worm detectors in all our tested environments—if it scanned at a rate no greater than 1 scan per 10 seconds.

7 Related Work

The most directly related work to ours—aside from the original publication of the detectors evaluated here—is a study by M. Patrick Collins and Michael K. Reiter that evaluates behavior-based (or payload-oblivious as they term it) detectors [30]. This work is closely tied to ours, but is complementary in nature. Their work, like ours, evaluates the effectiveness of several behavior-based detectors. The key distinction is that instead of monitoring an *internal* network for infections, they considered the performance of these systems in detecting incoming scanning from *external* networks. This is actually a substantially different problem than detecting internally infected hosts. There is a considerable volume of incoming scan traffic to most networks [31], and separating worm scanning from other scanning traffic is a different problem than detecting outgoing scans among legitimate outgoing traffic. They developed new metrics for their evaluation, measuring an attacker's payoff over an observable attack space. This new metric does not apply well to the job of detecting internal scanners, however, as the target address space of an internal scanner is potentially the entire IPv4 address space.

A work by Li, Salour, and Su surveys behavior and content-based worm detectors [32] and covers many of the works referenced here. They do not measure the performance of detectors, however, limiting their study to describing and classifying them instead. Our work briefly addresses broad classifications of detectors, but then focuses on their relative performance in real world situations.

8 Conclusions

The relative lack of worm attacks in recent years has caused network operators to focus their attention on other threats. However, Conficker and IKEE.B illustrate the continued threat that worms pose. Lapses in worm activity are not new—13 years separated the Morris worm from the series of large worm outbreaks in the early 2000's—and continued vigilance is required to protect our networks.

Despite the large number of worm detectors published, it is still unclear whether state-of-the-art systems are capable of coping with modern worms successfully. It is even unclear how these systems compare to each other. We have not seen a systematic comparison study that evaluates worm detectors against the same performance metrics across the same parameter values.

This paper addresses that issue. We focus on behavior-based worm detectors under different real-world environments, studying their false positive, false negative, and latency in detecting worms at various scanning rates using random, local-preference, or topological-aware scanning methods. We found that worms that scan at a low rate are the hardest to detect; for example, a topologically aware worm scanning one destination per minute can evade all tested detectors in all environments. Also, among all the environments we studied, the wireless environment poses the biggest challenge, where almost every detector incurs a lower—sometimes unacceptable—accuracy and higher latency than in other environments. No detector is a clear winner; TRW performs the best against the

random and local-preference worms, for example, but it fails badly at detecting a topologically aware worm.

References

1. Eisenberg, T., Gries, D., Hartmanis, J., Holcomb, D., Lynn, M.S., Santoro, T.: The Cornell commission: on Morris and the worm. Communications of the ACM 32(6), 706–709 (1989)
2. Moore, D., Shannon, C., Claffy, K.C.: Code-red: A case study on the spread and victims of an Internet worm. In: Proceedings of the ACM Internet Measurement Workshop, pp. 273–284 (2002)
3. Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: Inside the slammer worm. IEEE Security and Privacy 1(4), 33–39 (2003)
4. Symantec, I.: The downadup codex. Technical report, Symantec (March 2009)
5. Porras, P.A., Saidi, H., Yegneswaran, V.: An analysis of the ikee.b (duh) iPhone botnet. Technical report, SRI International (December 2009)
6. Sekar, V., Xie, Y., Reiter, M.K., Zhang, H.: A multi-resolution approach for worm detection and containment. In: Proceedings of the International Conference on Dependable Systems and Networks (2006)
7. Schechter, S.E., Jung, J., Berger, A.W.: Fast detection of scanning worm infections. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection (2004)
8. Gu, G., Sharif, M., Qin, X., Dagon, D., Lee, W., Riley, G.: Worm detection, early warning and response based on local victim information. In: Proceedings of the Annual Computer Security Applications Conference (2004)
9. Liang, Z., Sekar, R.: Fast and automated generation of attack signatures: A basis for building self-protecting servers. In: Proceedings of the Conference on Computer and Communications Security (2005)
10. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the Conference on Computer and Communications Security (2005)
11. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the Network and Distributed System Security Symposium (February 2005)
12. Tucek, J., Newsome, J., Lu, S., Huang, C., Xanthos, S., Brumley, D., Zhou, Y., Song, D.: Sweeper: A lightweight end-to-end system for defending against fast worms. In: Proceedings of the EuroSys Conference (2007)
13. Kim, H.A., Karp, B.: Autograph: Toward automated, distributed worm signature detection. In: Proceedings of the USENIX Security Symposium, pp. 271–286 (August 2004)
14. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Proceedings of the Symposium on Operating System Design and Implementation, pp. 45–60 (2004)
15. Wang, K., Cretu, G., Stolfo, S.J.: Anomalous payload-based worm detection and signature generation. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection (2005)
16. Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A content anomaly detector resistant to mimicry attack. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection (2006)

17. Li, Z., Wang, L., Chen, Y., Fu, Z.: Network-based and attack-resilient length signature generation for zero-day polymorphic worms. In: Proceedings of the IEEE International Conference on Network Protocols, pp. 164–173 (October 2007)
18. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: Proceedings of the IEEE Symposium on Security and Privacy (2005)
19. Mason, J., Small, S., Monrose, F., MacManus, G.: English shellcode. In: Proceedings of the Conference on Computer and Communications Security, pp. 524–533 (2009)
20. Jung, J., Milito, R., Paxson, V.: On the adaptive real-time detection of fast-propagating network worms. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment, pp. 175–192 (July 2007)
21. Collins, M.P., Reiter, M.K.: Hit-list worm detection and bot identification in large networks using protocol graphs. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection, pp. 276–295 (September 2007)
22. Wu, J., Vangala, S., Gao, L., Kwiat, K.: An effective architecture and algorithm for detecting worms with various scan techniques. In: Proceedings of the Network and Distributed System Security Symposium (2004)
23. Zou, C.C., Gong, W., Towsley, D., Gao, L.: The monitoring and early detection of Internet worms. ACM Transactions on Networking (2005)
24. Weaver, N., Staniford, S., Paxson, V.: Very fast containment of scanning worms. In: Proceedings of the USENIX Security Symposium, pp. 29–44 (2004)
25. DETER: Cyber defense technology experiment research (DETER) network, `http://www.isi.edu/deter/`
26. Stafford, S., Li, J., Ehrenkranz, T., Knickerbocker, P.: GLOWS: A high-fidelity worm simulator. Technical Report CIS-TR-2006-11, University of Oregon (2006)
27. LBNL/ICSI enterprise tracing project (2005), `http://www.icir.org/enterprise-tracing/`
28. Group, W.N.R.: WAND WITS: Auckland-IV trace data (April 2001), `http://wand.cs.waikato.ac.nz/wand/wits/auck/4/`
29. Umass trace repository, `http://traces.cs.umass.edu/`
30. Collins, M.P., Reiter, M.K.: On the limits of payload-oblivious network attack detection. In: Proceedings of the Symposium on Recent Advances in Intrusion Detection, pp. 251–270 (September 2008)
31. Allman, M., Paxson, V., Terrell, J.: A brief history of scanning. In: Proceedings of the ACM Internet Measurement Conference, pp. 77–82 (October 2007)
32. Li, P., Salour, M., Su, X.: A survey of internet worm detection and containment. IEEE Communications Society Surveys and Tutorials 10(1), 20–35 (2008)

Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams*

Liu Yang[1], Rezwana Karim[1], Vinod Ganapathy[1], and Randy Smith[2]

[1] Rutgers University
[2] Sandia National Laboratories

Abstract. Network intrusion detection systems (NIDS) make extensive use of regular expressions as attack signatures. Internally, NIDS represent and operate these signatures using finite automata. Existing representations of finite automata present a well-known time-space tradeoff: Deterministic automata (DFAs) provide fast matching but are memory intensive, while non-deterministic automata (NFAs) are space-efficient but are several orders of magnitude slower than DFAs. This time/space tradeoff has motivated much recent research, primarily with a focus on improving the space-efficiency of DFAs, often at the cost of reducing their performance.

This paper presents NFA-OBDDs, a symbolic representation of NFAs that retains their space-efficiency while improving their time-efficiency. Experiments using Snort HTTP and FTP signature sets show that an NFA-OBDD-based representation of regular expressions can outperform traditional NFAs by up to three orders of magnitude and is competitive with a variant of DFAs, while still remaining as compact as NFAs.

Keywords: NIDS, signature matching, ordered binary decision diagrams.

1 Introduction

Deep packet inspection allows network intrusion detection systems (NIDS) to accurately identify malicious traffic by matching the contents of network packets against attack signatures. In the past, attack signatures were keywords that could efficiently be matched using string matching algorithms. However, the increasing complexity of network attacks has lead the research community to investigate richer signature representations, which require the full power of regular expressions. Because NIDS are often deployed over high-speed network links, algorithms to match such rich signatures must also be efficient enough to provide high-throughput intrusion detection on large volumes of network traffic. This problem has spurred much recent research, and in particular has lead to the investigation of new representations of regular expressions that allow for efficient inspection of network traffic (*e.g.,* [1,2,3,4]).

To be useful for deep packet inspection in a NIDS, any representation of regular expressions must satisfy two key requirements: *time-efficiency* and *space-efficiency*. Time-efficiency requires the amount of time spent by the NIDS to process each byte

* Supported in part by NSF grants 0831268, 0915394, 0931992 and 0952128. L. Yang and R. Karim contributed equally, while R. Smith contributed while at the University of Wisconsin.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 58–78, 2010.

of network traffic to be small, thereby allowing large volumes of traffic to be matched quickly. Space-efficiency requires the size of the representation to be small, thereby ensuring that it will fit within the main memory of the NIDS. Space-efficiency also mandates that the size of the representation should grow proportionally (*e.g.,* linearly) with the number of attack signatures. This requirement is important because the increasing diversity of network attacks has lead to a quick growth in the number of signatures used by NIDS. For example, the number of signatures in Snort [5] has grown from 3,166 in 2003 to 15,047 in 2009.

Finite automata are a natural representation for regular expressions, but offer a tradeoff between time- and space-efficiency. Using deterministic finite automata (DFAs) to represent regular expressions allows efficient matching ($O(1)$ lookups to its transition table to process each input symbol), while non-deterministic finite automata (NFAs) can take up to $O(n)$ transition table lookups to process each input symbol, where n is the number of states in the NFA. However, NFAs are space-efficient, while DFAs for certain regular expressions can be exponentially larger than the corresponding NFAs [6]. More significantly, combining NFAs only leads to an additive increase in the number of states, while combining DFAs can result in a multiplicative increase, *i.e.,* an NFA that combines two NFAs with m and n states has up to $O(m + n)$ states, while a DFA that combines two DFAs with m and n states can have up to $O(m \times n)$ states. DFA representations for large sets of regular expressions often consume several gigabytes of memory, and do not fit within the main memory of most NIDS.

This time/space tradeoff has motivated much recent research, primarily with a focus on improving the space-efficiency of DFAs. These include heuristics to compress DFA transition tables (*e.g.,* [2,7]), techniques to combine regular expressions into multiple DFAs [4], and variable extended finite automata (XFAs) [3,8], which offer compact DFA representations and guarantee an additive increase in states when signatures are combined, provided that the regular expressions satisfy certain conditions. These techniques trade time for space, and though the resulting representations fit in main memory, their matching algorithms are slower than those for traditional DFAs.

In this paper, we take an alternative approach and instead focus on *improving the time-efficiency of NFAs.* NFAs are not currently in common use for deep packet inspection, and understandably so—their performance can be several orders of magnitude slower than DFAs. Nevertheless, NFAs offer a number of advantages over DFAs, and we believe that further research on improving their time-efficiency can make them a viable alternative to DFAs. Our position is supported in part by these observations:

- **NFAs are more compact than DFAs.** Determinizing an NFA involves a subset construction algorithm, which can result in a DFA with exponentially more states than an equivalent NFA [6].
- **NFA combination is space-efficient.** Combining two NFAs simply involves linking their start states together by adding new ϵ transitions; the combined NFA is therefore only as large as the two NFAs put together. This feature of NFAs is particularly important, given that the diversity of network attacks has pushed NIDS vendors to deploy an ever increasing number of signatures. In contrast, combining two DFAs can result in a multiplicative increase in the number of states, and the combined DFA may be much larger than its constituent DFAs.

- **NFAs can readily be parallelized.** An NFA can be in a set of states (called the *frontier*) at any instant during its operation, each of which may contain multiple outgoing transitions for an input symbol. States in the frontier can be processed in parallel as new input symbols are encountered [9,10].

Motivated by these advantages, we develop a new approach to improve the time-efficiency of NFAs. The frontier of an NFA can contain $O(n)$ states, each of which must be processed using the NFA's transition relation for each input symbol to compute a new frontier, thereby resulting in slow operation. Although this frontier can be processed in parallel to improve performance, NFAs for large signature sets may contain several thousand states in their frontier at any instant. Commodity hardware is not yet well-equipped to process such large frontiers in parallel.

Our core insight is that a technique to efficiently apply an NFA's transition relation to a *set of states* can greatly improve the time-efficiency of NFAs. Such a technique would apply the transition relation to all states in the frontier in a single operation to produce a new frontier. We develop an approach that uses *Ordered Binary Decision Diagrams* [11] (OBDDs) to implement such a technique. Our use of OBDDs to process NFA frontiers is inspired by symbolic model checking, where the use of OBDDs allows the verification of systems that contain an astronomical number of states [12]. NFAs that use OBDDs (**NFA-OBDDs**) can be constructed from regular expressions in a fully automated way, and are robust in the face of structural complexities in these regular expressions (*e.g.,* counters [8, Section 6.2]).

To evaluate the feasibility of our approach, we constructed NFAs in software using HTTP and FTP signatures from Snort. We operated these NFAs using OBDDs and evaluated their time-efficiency and space-efficiency using HTTP and FTP traffic obtained from our department's network. Our experiments showed that NFA-OBDDs outperform traditional NFAs by approximately *three orders of magnitude*—about $1645\times$ in the best case. Our experiments also showed that NFA-OBDDs retain the space-efficiency of NFAs. In contrast, our machine ran out of memory when trying to construct DFAs (or their variants) from our signature sets.

Our main contributions are as follows:

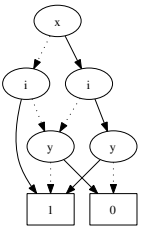
- **Design of NFA-OBDDs.** We develop a novel technique that uses OBDDs to improve the time-efficiency of NFAs (Section 3). We also describe how NFA-OBDDs can be used to improve the time and space-efficiency of NFA-based multi-byte matching (Section 5).
- **Comprehensive evaluation using Snort signatures.** We evaluated NFA-OBDDs using Snort's HTTP and FTP signature sets and observed a speedup of about three orders of magnitude over traditional NFAs. We also compared the performance of NFA-OBDDs against a variety of automata implementations, including the PCRE package and a variant of DFAs (Section 4).

2 Ordered Binary Decision Diagrams

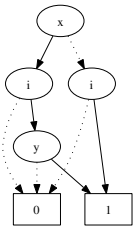
An OBDD is a data structure that can represent arbitrary Boolean formulae. OBDDs transform Boolean function manipulation into efficient graph transformations, and have found wide use in a number of application domains. For example, OBDDs are used

x	*i*	*y*	*f(x, i, y)*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

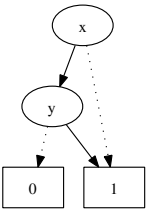
(a) A Boolean function $f(x, i, y)$.



(b) OBDD(f) with $x < i < y$.



(c) APPLY($\wedge$, OBDD(f), OBDD($I(i)$)).



(d) RESTRICT(OBDD(f), $i \leftarrow 1$**).**

Fig. 1. An example of a Boolean formula, its OBDD, and various operations on the OBDD. Solid edges are labeled 1, dotted edges are labeled 0.

extensively by model checkers to improve the efficiency of state-space exploration algorithms [12]. In this section, we present an informal overview of OBDDs, and refer interested readers to Bryant's seminal article [11] for details.

An OBDD represents a Boolean function $f(x_1, x_2, \ldots, x_n)$ as a rooted, directed acyclic graph (DAG). The DAG has two *terminal* nodes, which are labeled **0** and **1**, and have no outgoing edges. Each remaining *non-terminal* node is associated with a label $\in \{x_1, x_2, \ldots, x_n\}$, and has two outgoing edges labeled **0** and **1**. An OBDD is ordered in the sense that node labels are associated with a total order $<$. Node labels along all paths in the OBDD from the root to the terminal nodes follow this total order.[1] To evaluate the Boolean formula denoted by an OBDD, it suffices to traverse appropriately labeled edges from the root to the terminal nodes of the DAG. Figure 1(b) depicts an example of an OBDD for the Boolean formula $f(x, i, y)$ shown in Figure 1(a) with the variable ordering $x < i < y$.

OBDDs allow Boolean functions to be manipulated efficiently. With OBDDs, checking the satisfiability (or unsatisfiability) of a Boolean formula is a constant time operation, because it suffices to check whether the terminal node labeled **1** (respectively, **0**) is present in the OBDD. The APPLY and RESTRICT operations [11], described below, allow OBDDs to be combined and modified with a number of Boolean operators. These two operations are implemented as a series of graph transformations and reductions to the input OBDDs, and have efficient implementations (their time complexity is polynomial in the size of the input OBDDs).

APPLY allows binary Boolean operators, such as $\wedge$ and $\vee$, to be applied to a pair of OBDDs. The two input OBDDs, $\mathsf{OBDD}(f)$ and $\mathsf{OBDD}(g)$, must have the same variable ordering. APPLY(OP, $\mathsf{OBDD}(f)$, $\mathsf{OBDD}(g)$) computes $\mathsf{OBDD}(f \text{ OP } g)$, which has the same variable ordering as the input OBDDs. Figure 1(c) presents the OBDD obtained by combining the OBDD in Figure 1(b) with $\mathsf{OBDD}(I(i))$, where I is the identity function. The RESTRICT operation is unary, and produces as output an OBDD in which the values of some of the variables of the input OBDD have been fixed to a certain value. That is, RESTRICT($\mathsf{OBDD}(f)$, $x \leftarrow k$) = $\mathsf{OBDD}(f|_{(x \leftarrow k)})$, where $f|_{(x \leftarrow k)}$ denotes that x is assigned the value k in f. In this case, the output OBDD does not have

[1] DAGs denoting OBDDs satisfy additional properties, as described in Bryant's article. However, they are not directly relevant for this discussion, and we elide them for brevity.

any nodes with the label x. Figure 1(d) shows the OBDD obtained as the output of RESTRICT(OBDD(f), $i \leftarrow 1$), where OBDD(f) is the OBDD of Figure 1(b).

APPLY and RESTRICT can be used to implement existential quantification, which is used in a key way in the operation of NFA-OBDDs, as described in Section 3. In particular, $\exists x_i.f(x_1,\ldots,x_n) = f(x_1,\ldots,x_n)|_{(x_i \leftarrow 0)} \vee f(x_1,\ldots,x_n)|_{(x_i \leftarrow 1)}$. Therefore, we have: OBDD($\exists\, x_i.f(x_1,\ldots,x_n)$) = APPLY($\vee$, RESTRICT(OBDD($f$), $x_i \leftarrow 1$), RESTRICT(OBDD(f), $x_i \leftarrow 0$)). Note that OBDD($\exists\, x_i.f(x_1,\ldots,x_n)$) will not have a node labeled x_i.

Representing Relations and Sets. OBDDs can be used to represent relations of arbitrary arity. If R is an n-ary relation over the domain $\{0, 1\}$, then we define its *characteristic function* f_R as follows: $f_R(x_1,\ldots,x_n) = 1$ if and only if $R(x_1,\ldots,x_n)$. For example, the characteristic function of the 3-ary relation $R = \{(1,0,1),(1,1,0)\}$ is $f_R(x_1,x_2,x_3) = (x_1 \wedge \bar{x_2} \wedge x_3) \vee (x_1 \wedge x_2 \wedge \bar{x_3})$. f_R is a Boolean function and can therefore be expressed using an OBDD.

An n-ary relation Q over an arbitrary domain D can be similarly expressed using OBDDs by *bit-blasting* each of its elements. That is, if the domain D has m elements, we map each of its elements uniquely to bit-strings containing $\lceil \lg m \rceil$ bits (call this mapping ϕ). We then define a new relation $R(\phi(x_1),\ldots,\phi(x_n)) = Q(x_1,\ldots,x_n)$. R is a $n \times \lceil \lg m \rceil$-ary relation over $\{0,1\}$, and can be converted into an OBDD using its characteristic function.

A set of elements over an arbitrary domain D can also be expressed as an OBDD because sets are unary relations, *i.e.,* if S is a set of elements over a domain D, then we can define a relation R_S such that $R_S(s) = 1$ if and only if $s \in S$. Operations on sets can then be expressed as Boolean operations and performed on the OBDDs representing these sets. For example, $S \subseteq T$ can be implemented as OBDD(S) $\longrightarrow$ OBDD(T) (logical implication), while ISEMPTY($S \cap T$) is equivalent to checking whether OBDD(S) $\wedge$ OBDD(T) is satisfiable. The conversion of relations and sets into OBDDs is used in a key way in the construction and operation of NFA-OBDDs, which we describe next.

3 Representing and Operating NFAs

We represent an NFA using a 5-tuple: (Q, Σ, Δ, q_0, *Fin*), where Q is a finite set of states, Σ is a finite set of input symbols (the alphabet), Δ: $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ is a transition function, $q_0 \in Q$ is a start state, and *Fin* $\subseteq Q$ is a set of accepting (or final) states. The transition function $\Delta(s, i) = T$ describes the set of all states $t \in T$ such that there is a transition labeled i from s to t. Note that Δ can also be expressed as a relation δ: $Q \times \Sigma \times Q$, so that $(s,i,t) \in \delta$ for all $t \in T$ such that $\Delta(s,i) = T$. We will henceforth use δ to denote the set of transitions in the NFA.

An NFA may have multiple outgoing transitions with the same input symbol from each state. Hence, it maintains a *frontier* F of states that it can currently be in. The frontier is initially the singleton set $\{q_0\}$ but may include any subset of Q during the operation of the NFA. For each symbol in the input string, the NFA must process all of the states in F and find a new set of states by applying the transition relation.

While non-determinism leads to frontiers of size $O(|Q|)$ in NFAs, it also makes them space-efficient in two ways. First, NFAs for certain regular expressions are exponentially smaller than the corresponding DFAs, *e.g.,* an NFA for `(0|1)*1(0|1)`^{n} has $O(n)$ states, while the corresponding DFA has $O(2^n)$ states [6]. Second, and perhaps more significant from the perspective of NIDS, NFAs can be combined space-efficiently while DFAs cannot. To combine a pair of NFAs, NFA_1 and NFA_2, it suffices to create a new state q_{new}, add ϵ transitions from q_{new} to the start states of NFA_1 and NFA_2, and designate q_{new} to be the start state of the combined NFA. This leads to an NFA with $O(|Q_1| + |Q_2|)$ states. In contrast, combining two DFAs, DFA_1 and DFA_2, results in a multiplicative increase in the number of states because the combined DFA must have a state corresponding to $s \times t$ for each pair of states s and t in DFA_1 and DFA_2, respectively. The number of states in the DFA can possibly be reduced using minimization, but this does not always help. For example, the DFAs for the regular expressions `ab*cd*` and `ef*gh*` have 5 states and 6 transitions each, and the combined DFA (minimized) has 16 states and 22 transitions.

NFA Operation using Boolean Functions. We now describe how the process of applying an NFA's transition relation to a frontier of states can be expressed as a sequence of Boolean function manipulations. NFA-OBDDs implement Boolean functions and operations on them using BDDs. For the discussion below and in the rest of this paper, we assume NFAs in which ϵ transitions have been eliminated (using standard techniques [6]). This is mainly for ease of exposition; NFAs with ϵ transitions can also be expressed using NFA-OBDDs. Note that ϵ elimination may increase the total number of transitions in the NFA, but does not increase the number of states.

We now define four Boolean functions for an NFA (Q, Σ, δ, q_0, *Fin*). These functions use three vectors of Boolean variables: $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{i}$. The vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are used to denote states in Q, and therefore contain $\lceil \lg |Q| \rceil$ variables each. The vector $\boldsymbol{i}$ denotes symbols in Σ, and contains $\lceil \lg |\Sigma| \rceil$ variables. As an example, for the NFA in Figure 2, these vectors contain one Boolean variable each; we denote them as x, y, and i.

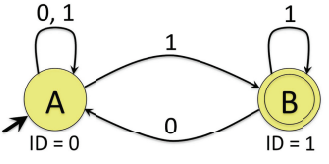


Fig. 2. NFA for `(0|1)*1`

- $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ denotes the NFA's transition relation δ. Recall that δ is a set of triples (s, i, t), such that there is a transition labeled i from state s to state t. It can therefore be represented as a Boolean function as described in Section 2. For example, consider the NFA in Figure 2. Using 0 to denote state A and 1 to denote state B, $\mathcal{T}(x, i, y)$ is the function shown in shown in Figure 1(a).
- $\mathcal{I}_\sigma(\boldsymbol{i})$ is defined for each $\sigma \in \Sigma$, and denotes a Boolean representation of that symbol. For the NFA in Figure 2, $\mathcal{I}_0(i) = \bar{i}$ (*i.e.,* $i = 0$) and $\mathcal{I}_1(i) = i$.
- $\mathcal{F}(\boldsymbol{x})$ denotes the current set of frontier states of the NFA. It is thus a Boolean representation of the set F at any instant during the operation of the NFA. For the example in Figure 2, if $F = \{\text{A}\}$, $\mathcal{F}(x) = \bar{x}$, while if $F = \{\text{A, B}\}$, then $\mathcal{F}(x) = x \vee \bar{x}$.
- $\mathcal{A}(\boldsymbol{x})$ is a Boolean representation of *Fin*, and denotes the accepting states. In Figure 2, $\mathcal{A}(x) = x$.

Note that $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$, $\mathcal{I}_\sigma(\boldsymbol{i})$ and $\mathcal{A}(\boldsymbol{x})$ can be computed automatically from any representation of NFAs. The initial frontier $F = \{q_0\}$ can also be represented as a Boolean formula. Suppose that the frontier at some instant during the operation of the NFA is $\mathcal{F}(\boldsymbol{x})$, and that the next symbol in the input is σ. The following Boolean formula, $\mathcal{G}(y)$, symbolically denotes the new frontier of states in the NFA after σ has been processed.

$$\mathcal{G}(\boldsymbol{y}) = \exists\, \boldsymbol{x}.\exists\, \boldsymbol{i}.[\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}) \wedge \mathcal{I}_\sigma(\boldsymbol{i}) \wedge \mathcal{F}(\boldsymbol{x})]$$

To see why $\mathcal{G}(\boldsymbol{y})$ is the new frontier, consider the truth table of the Boolean function $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$. By construction, this function evaluates to 1 only for those values of $\boldsymbol{x}$, $\boldsymbol{i}$, and $\boldsymbol{y}$ for which $(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ is a transition in the automaton. Similarly, the function $\mathcal{F}(\boldsymbol{x})$ evaluates to 1 only for the values of $\boldsymbol{x}$ that denote states in the current frontier of the NFA. Thus, the conjunction of $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ with $\mathcal{F}(\boldsymbol{x})$ and $\mathcal{I}_\sigma(\boldsymbol{i})$ only "selects" those rows in the truth table of $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ that correspond to the outgoing transitions from states in the frontier labeled with the symbol σ. However, the resulting conjunction is a Boolean formula in $\boldsymbol{x}$, $\boldsymbol{i}$ and $\boldsymbol{y}$. To find the new frontier of states, we are only interested in the values of $\boldsymbol{y}$ (*i.e.,* the target states of the transitions) for which the conjunction has a satisfying assignment. We achieve this by existentially quantifying $\boldsymbol{x}$ and $\boldsymbol{i}$ to obtain $\mathcal{G}(\boldsymbol{y})$. To express the new frontier in terms of the Boolean variables in $\boldsymbol{x}$, we rename the variables in $\boldsymbol{y}$ with the corresponding ones in $\boldsymbol{x}$.

We illustrate this idea using the example in Figure 2. Suppose that the current frontier of the NFA is $F = \{\mathrm{A}, \mathrm{B}\}$, and that the next input symbol is a 0, which causes the new frontier to become $\{\mathrm{A}\}$. In this case, $\mathcal{T}(x, i, y)$ is the function shown in Figure 1(a), $\mathcal{I}_0(i) = \bar{i}$ and $\mathcal{F}(x) = x \vee \bar{x}$. We have $\mathcal{T}(x, i, y) \wedge \mathcal{I}_0(i) \wedge \mathcal{F}(x) = (x \wedge \bar{i} \wedge \bar{y})$. Existentially quantifying x and i from the result of this conjunction, we get $\mathcal{G}(y) = \bar{y}$. Renaming the variable y to x, we get $\mathcal{F}(x) = \bar{x}$, which is a Boolean formula that denotes $\{\mathrm{A}\}$, the new frontier.

To determine whether the NFA accepts an input string, it suffices to check that $F \cap \mathit{Fin} \neq \emptyset$. Using the Boolean notation, this translates to check whether $\mathcal{F}(\boldsymbol{x}) \wedge \mathcal{A}(\boldsymbol{x})$ has a satisfying assignment. In the example above with $F = \{\mathrm{A}\}$, $\mathcal{F}(x) = \bar{x}$ and $\mathcal{A}(x) = x$, so the NFA is not in an accepting configuration. Recall that checking satisfiability of a Boolean function is an $O(1)$ operation if the function is represented as an OBDD.

NFA-OBDDs. The main idea behind NFA-OBDDs is to represent and manipulate the Boolean functions discussed above using OBDDs. Formally, an NFA-OBDD for an NFA $(Q, \Sigma, \delta, q_0, \mathit{Fin})$ is a 7-tuple $(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \mathsf{OBDD}(\mathcal{T}), \{\mathsf{OBDD}(\mathcal{I}_\sigma \mid \forall \sigma \in \Sigma)\}, \mathsf{OBDD}(\mathcal{F}_{q_0}), \mathsf{OBDD}(\mathcal{A}))$, where $\boldsymbol{x}$, $\boldsymbol{i}$, $\boldsymbol{y}$ are vectors of Boolean variables, and $\mathcal{T}$, $\mathcal{I}_\sigma$, and $\mathcal{A}$ are the Boolean formulae discussed earlier. $\mathcal{F}_{q_0}$ denotes the Boolean function that denotes the frontier $\{q_0\}$. For each input symbol σ, the NFA-OBDD obtains a new frontier as discussed earlier. The main difference is that the Boolean operations are performed as operations on OBDDs.

The use of OBDDs allows NFA-OBDDs to be more time-efficient than NFAs. In an NFA, the transition table must be consulted for each state in the frontier, leading to $O(|\delta| \times |F|)$ operations per input symbol. In contrast, the complexity of OBDD operations to obtain a new frontier is $O(\textsc{sizeof}(\mathsf{OBDD}(\mathcal{T})) \times \textsc{sizeof}(\mathsf{OBDD}(\mathcal{F})))$. Because OBDDs are a compact representation of the frontier F and the transition

relation δ, NFA-OBDDs are more time-efficient than NFAs. The improved performance of NFA-OBDDs is particularly pronounced when the transition table of the NFA is sparse or the NFA has large frontiers. This is because OBDDs can effectively remove redundancy in the represention of δ and F.

NFA-OBDDs retain the space-efficiency of NFAs because NFA-OBDDs can be combined using the same algorithms that are used to combine NFAs. Although the use of OBDDs may lead NFA-OBDDs to consume more memory than NFAs, our experiments show that the increase is marginal. In particular, the cost is dominated by $\mathsf{OBDD}(\mathcal{T})$, which has a total of $2 \times \lceil \lg |Q| \rceil + \lceil \lg |\Sigma| \rceil$ Boolean variables. Even in the worst case, this OBDD consumes only $O(|Q|^2 \times |\Sigma|)$ space, which is comparable to the worst-case memory consumption of the transition table of a traditional NFA. However, in practice, the memory consumption of NFA-OBDDs is much smaller than this asymptotic limit.

4 Implementation and Evaluation

We evaluated the feasibility of our approach using a software-based implementation of NFA-OBDDs. As depicted in Figure 3, the implementation consists of two offline components and an online component.

The offline components are executed once for each set of regular expressions, and consist of re2nfa and nfa2obdd. The re2nfa component accepts a set of regular expressions as input, and produces an ϵ-free NFA as output. To do so, it first constructs NFAs for each of the regular expressions using Thompson's construction [13,6], combines these NFAs into a single NFA, and eliminates ϵ transitions. The nfa2obdd component analyzes this NFA to determine the number of Boolean variables needed (*i.e.,* the sizes of the $\boldsymbol{x}$, $\boldsymbol{i}$ and $\boldsymbol{y}$ vectors), and constructs $\mathsf{OBDD}(\mathcal{T})$, $\mathsf{OBDD}(\mathcal{A})$, $\mathsf{OBDD}(\mathcal{I}_\sigma)$ for each $\sigma \in \Sigma$, and $\mathsf{OBDD}(\mathcal{F}_{q_0})$.

It is well-known that the size of an OBDD for a Boolean formula is sensitive to the total order imposed on its variables. For the NFA-OBDDs used in our experiments, we empirically determined that an ordering of variables of the form $\boldsymbol{i} < \boldsymbol{x} < \boldsymbol{y}$ yields the best performance for NFA-OBDDs. For example, we found that an implementation of NFA-OBDDs that uses the variable ordering $\boldsymbol{x} < \boldsymbol{i} < \boldsymbol{y}$ operates more than an order of magnitude slower than one that uses the ordering $\boldsymbol{i} < \boldsymbol{x} < \boldsymbol{y}$; we therefore used the latter ordering in our implementation. Within each vector, nfa2obdd uses a simple sorting scheme to order variables. Although it is NP-hard to choose a total order that yields the most compact OBDD for a Boolean function [11], future work could develop heuristics that leverage the structure of the input regular expressions to determine orderings that yield high-performance NFA-OBDDs.

The online component, exec_nfaobdd, begins execution by reading these OBDDs into memory and processes a stream of network packets. It matches the payloads of these network packets against the regular expressions using the NFA-OBDD. To manipulate OBDDs and produce a new frontier for each input symbol processed, this component interfaces with Cudd, a popular C++-based OBDD library [14]. It checks whether each frontier $\mathcal{F}$ produced during the operation of the NFA-OBDD contains an accepting state. If so, it emits a warning with the offset of the character in the input stream that triggered a match, as well as the regular expression(s) that matched the input. Note

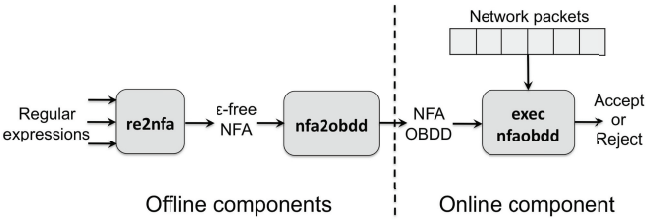


Fig. 3. Components of our software-based implementation of NFA-OBDDs

that in a NIDS setting, it is important to check whether the frontier $\mathcal{F}$ obtained after processing *each* input symbol contains an accepting state. This is because *any* byte in the network input may cause a transition in the NFA that triggers a match with a regular expression. We call this the *streaming* model because the NFA continuously processes input symbols from a network stream.

Data Sets. We evaluated our implementation of NFA-OBDDs with three sets of regular expressions [15]. The first set was obtained from the authors of the XFA paper [8], and contains 1503 regular expressions that were synthesized from the March 2007 snapshot of the Snort HTTP signature set. The second and third sets, numbering 2612 and 98 regular expressions, were obtained from the October 2009 snapshot of the Snort HTTP and FTP signature sets, respectively. About 50% of these regular expressions were taken from the `uricontent` fields of the signatures, while the rest were extracted from the `pcre` fields. Although extracting just `pcre` fields from individual Snort rules only captures a portion of the corresponding signatures, it suffices for our experiments, because our primary goal is to evaluate the performance of NFA-OBDDs against other regular-expression based techniques. All three sets of regular expressions include client-side and server-side signatures. For all sets, we excluded Snort signatures that contained non-regular constructs, such as back-references and subroutines (which are allowed by PCRE [16]), because these constructs cannot be implemented in NFA-based models. In all, we excluded 1837 HTTP and 41 FTP signatures due to non-regular constructs.

To evaluate the performance of HTTP signatures, we fed traces of live HTTP traffic obtained from our department's network to exec_nfaobdd. We collected this traffic over a one week period in August 2009. This traffic was collected using tcpdump, and includes whole packets of port 80 traffic from our departmental Web server and our lab's machines.

The traffic observed during this period consisted largely of Web traffic typically observed at an academic department's main Web server; most of the traffic was to view and query Web pages hosted by the department. Overall, we observed connections from 18,618 distinct source IP addresses during this period, with 653,670 GET, 137,737 POST, 3,504 HEAD, and 1,576 PUT commands. This traffic triggered 1,816,410 and 17,107,588 matches in the HTTP/1503 and HTTP/2612 signature sets, corresponding to 47 and 120 distinct signatures, respectively.[2] The payloads in these packets ranged

[2] These numbers are not indicative of the number of alerts produced by Snort because our signature sets only contain patterns from the `pcre` and `uricontent` fields of the Snort rules. The large number of matches is because signatures contained patterns common in HTTP packets.

in size from 1 byte to 1460 bytes, with an average of 126 bytes (standard deviation of 271). However, we partitioned this traffic into 33 traces of various sizes, containing between 5.1MB–1.24GB worth of data. We did so because the the NFA and PCRE-based implementations discussed in this section were too slow to process the weeklong traffic trace. The size distribution of these traces was as follows: 21 traces of 5.1-7.2MB, 9 traces of 10.3-20.1MB, and one trace each of 227.2MB, 575.8MB, and 1.24GB.

We evaluated the FTP signatures using two traces of live FTP traffic (from the command channel), obtained over a two week period in March 2010 from our department's FTP server; these FTP traces contained 19.4MB and 24.7MB worth of data. The traffic consisted of FTP requests to fetch and update technical reports hosted by our department. We observed traffic from 528 distinct source IP addresses during this period. Statistics on various FTP commands observed during this period appear in the table below (commands that were not observed are not reported). This traffic triggered 9,656 and 15,976 matches in the FTP/98 signature set, corresponding to matches on 6 and 5 distinct signatures, respectively. The payload sizes of packets ranged from 2 to 402 bytes with an average of 40 bytes (standard deviation of 44).

Command	CWD	LIST	MDTM	MKD	PASS	PORT	PWD	QUIT	RETR	SIZE	STOR	TYPE	USER
Number	62,561	3,098	613	89	14,701	232	453	12,244	7,676	1,110	1,401	12,201	14,834

We also used synthetic traces during our experiments, but do not report these results in the paper because they are substantially similar to those obtained using real traffic. Because our primary goal is to study the performance of NFA-OBDDs, we assume that the network traces have been processed using standard NIDS operations, such as defragmentation and normalization. We fed these traces, which were in tcpdump format, to exec_nfaobdd.

Experimental Setup. All our experiments were performed on a Intel Core2 Duo E7500 Linux-2.6.27 machine, running at 2.93GHz with 2GB of memory (however, our programs are single-threaded, and only used one of the available cores). We used the Linux `/proc` file system to measure the memory consumption of nfa2obdd and the Cudd ReadMemoryInUse utility to obtain the memory consumption of exec_nfaobdd. We instrumented both these programs to report their execution time using processor performance counters. We report the performance of exec_nfaobdd as the number of CPU cycles to process each byte of network traffic (cycles/byte), *i.e.,* fewer processing cycles/byte imply greater time-efficiency. All our implementations were in C++; we used the GNU `g++` compiler suite (v4.3.2) with the O6 optimization level to produce the executables used for experimentation.

Our experiments show that NFA-OBDDs: (**1**) outperform traditional NFAs by up to three orders of magnitude while retaining their space-efficiency; (**2**) outperform or are competitive in performance with the PCRE package, a popular library for regular expression matching; (**3**) are competitive in performance with variants of DFAs while being drastically less memory-intensive.

Constructing NFA-OBDDs. We used nfa2obdd to construct NFA-OBDDs from ϵ-free NFAs of the regular expression sets. Figure 4 presents statistics on the sizes of the input NFAs, the size of the largest of the four OBDDs in the NFA-OBDD (OBDD($\mathcal{T}$)),

Signature Set	#Reg. Exps.	Size of the input NFA		|OBDD($\mathcal{T}$)|	Construction
		#States	#Transitions	#Nodes	Time/Memory
HTTP (March 2007)	1503	159,734	3,986,769	659,981	305sec/176MB
HTTP (October 2009)	2612	239,890	5,833,911	989,236	453sec/176MB
FTP (October 2009)	98	26,536	5,927,465	69,619	246sec/134MB

Fig. 4. NFA-OBDD construction results

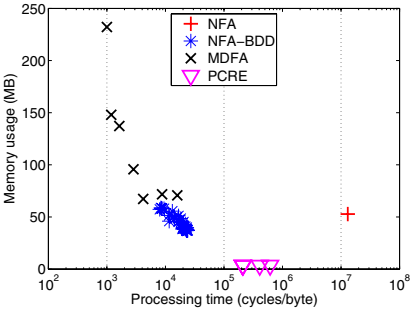
and the time taken and memory consumed by nfa2obdd. For the NFA-OBDDs corresponding to the HTTP signature sets, the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ had 18 Boolean variables each, while the vector $\boldsymbol{i}$ had 8 Boolean variables to denote the 256 possible ASCII characters. For the NFA-OBDD corresponding to the FTP signature set, the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ had 15 Boolean variables each. We also tried to determinize these NFAs to produce DFAs, but the determinizer ran out of memory in all three cases.

Performance of NFA-OBDDs. Figure 5 depicts the performance of NFA-OBDDs. Figures 5(a) and 5(b) show the performance for each of the 33 HTTP traces, while Figure 5(c) shows the performance for both FTP traces. Figure 5(d) also presents the raw throughput and memory consumption of NFA-OBDDs observed for each signature set. The throughput and memory consumption of NFA-OBDDs varies across different traces for each signature set; this variance can be attributed to the size and shapes of OBDD($\mathcal{F}$) (the OBDD of the NFA's frontier) observed during execution. We also observed that larger traces are processed *more* efficiently on average than smaller traces. For example, in Figure 5(a), the 1.24GB trace was processed at 7,935 cycles/byte, whereas a 20MB trace was processed at 19,289 cycles/byte. We hypothesize that the improved throughput observed for larger traces is because of cache effects. As exec_nfaobdd executes, it is likely that NFA-OBDDs that are frequently observed will be cached, therefore producing improved throughput for larger traces.

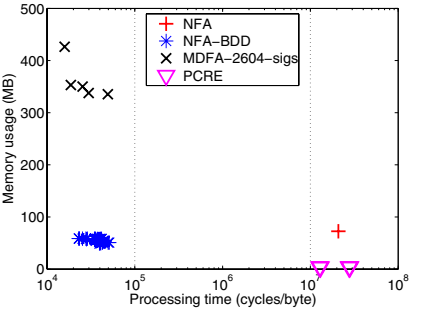
Comparison with NFAs. We compared the performance of NFA-OBDDs with an implementation of NFAs that uses Thompson's algorithm. This algorithm maintains a frontier F, and operates as follows: for each state s in the frontier F, fetch the set of targets T_s of the transitions labeled σ and compute the new frontier $F' = \bigcup_{s \in F} T_s$.

Our implementation of NFAs makes heavy use of the C++ standard template library. It stores the transition table as an array of $|Q|$ multimaps. The entry for state s denotes the set of outgoing transitions from s, where each transition is of the form (σ, t). There may be multiple entries with the same input symbol σ in each multimap, corresponding to all the states reachable from s via transitions labeled σ. The performance and memory consumption of our NFA implementation was relatively stable across the traces used for each signature set. Figure 5 therefore reports only the averages across these traces.

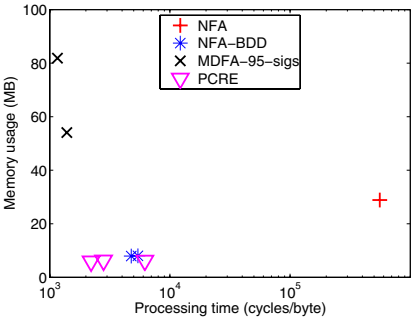
As Figure 5 shows, NFA-OBDDs outperform NFAs for all three sets of signatures by approximately three orders of magnitude for the HTTP signatures, and two orders of magnitude for the FTP signatures. In Figure 5(a), for example, NFA-OBDDs are between 570×–1645× faster than NFAs, while consuming approximately the same amount of memory. The difference in the performance gap between NFA-OBDDs and NFAs for the HTTP and FTP signatures can be attributed to the number and structure of these signatures. As discussed in Section 3, the benefits of NFA-OBDDs are more



(a) HTTP/1503 regular expressions



(b) HTTP/2612 regular expressions



(c) FTP/98 regular expressions

Sig. Set	Processing time	Memory
NFA-OBDDs		
HTTP/1503	7,935–22,895 cycles/byte	39–59MB
HTTP/2612	22,968–51,215 cycles/byte	54–61MB
FTP/98	5,095 cycles/byte	8MB
NFAs		
HTTP/1503	1.3×10^7 cycles/byte	53MB
HTTP/2612	2.1×10^7 cycles/byte	73MB
FTP/98	5.6×10^5 cycles/byte	29MB
PCRE		
HTTP/1503	$2.1\text{–}6.2 \times 10^5$ cycles/byte	3.6MB
HTTP/2612	$1.3\text{–}2.8 \times 10^7$ cycles/byte	3.9MB
FTP/98	2,210–6,185 cycles/byte	5.9–6.2MB
MDFA (partial signature sets in Figure 5(b) and (c))		
HTTP/1503	1,000–15,951 cycles/byte	71–232MB
HTTP/2604	15,891–49,296 cycles/byte	335–426MB
FTP/95	1,160–1,386 cycles/byte	54–82MB

(d) Raw performance numbers

Fig. 5. Comparing memory versus processing time of (1) NFA-OBDDs, (2) traditional NFAs, (3) the PCRE package, and (4) different MDFAs for the Snort HTTP and FTP signature sets. The x-axis is in log-scale. Note that Figure 5(b) and Figure 5(c) only report the performance of MDFAs with 2604 and 95 regular expressions, respectively.

pronounced if larger frontiers are to be processed. Since there are a larger number of HTTP signatures, the frontier for the corresponding NFAs are larger. As a result, NFA-OBDDs are much faster than the corresponding NFAs for HTTP signatures than for FTP signatures. Nevertheless, *these results clearly demonstrate that OBDDs can improve the time-efficiency of NFAs without compromising their space-efficiency.*

Comparison with the PCRE package. We compared the performance of NFA-OBDDs with the PCRE package used by a number of tools, including Snort and Perl. The PCRE package represents regular expressions using a tree structure, and matches input strings against this structure using a backtracking algorithm. For a given input string, this algorithm iteratively explores paths in the tree until it finds an accepting state. If it fails to find an accepting state in one path, it backtracks and tries another path until all paths have been exhausted.

Figure 5 reports three numbers for the performance of the PCRE package, corresponding to different values of configuration parameters of the package. In both

Figure 5(a) and (b), NFA-OBDDs outperform the PCRE package. The throughput of NFA-OBDDs is about an order of magnitude ($9\times$–$26\times$) better than the fastest configuration of the PCRE package for the set HTTP/1503. The difference in performance is more pronounced for the set HTTP/2612, where NFA-OBDDs outperform the most time-efficient PCRE configuration by $248\times$–$554\times$. The poorer throughput of the PCRE package for the second set of signatures is likely because the backtracking algorithm that it employs degrades in performance as the number of paths to be explored in the NFA increases. However, in both cases, the PCRE package is more space-efficient than NFA-OBDDs, and consumes between 3.7MB–4MB memory.

For the FTP signatures (Figure 5(c)), NFA-OBDDs are about $2\times$ slower than the fastest PCRE configuration. However, unlike NFA-OBDDs which report all substrings of an input packet that match signatures, this PCRE configuration only reports the first matching substring. The performance of the PCRE configurations that report all matching substrings is comparable to that of NFA-OBDDs.

Note that in all cases, the PCRE package outperforms our NFA implementation, which use Thompson's algorithm [13] to parse input strings. Despite this gap in performance, Cox [17] shows that Thompson's algorithm performs more consistently than the backtracking approach employed by PCRE. For example, the backtracking approach is vulnerable to algorithmic complexity attacks, where a maliciously-crafted input can trigger the worst-case performance of the algorithm [18].

Comparison with DFA variants. We compared the performance of NFA-OBDDs with a variant of DFAs, called multiple DFAs (MDFAs), produced by set-splitting [4].[3] An MDFA is a collection of DFAs representing a set of regular expressions. Each DFA represents a disjoint subset of the regular expressions. To match an input string against an MDFA, each constituent DFA is simulated against the input string to determine whether there is a match. MDFAs are more compact than DFAs because they result in a less than multiplicative increase in the number of states. However, MDFAs are also slower than DFAs because all the constituent DFAs must be matched against the input string. An MDFA that has a larger number of constituent DFAs will be more compact, but will also have lower time-efficiency than an MDFA with fewer DFAs.

Using Yu *et al.*'s algorithms [4], we produced several MDFAs by combining the Snort signatures in several ways, each with different space/time utilization. Each point in Figure 5 denotes the performance of *one* MDFA (again, averaged over all the input traces), which in turn consists of a collection of combined DFAs as described above.

Producing MDFAs for the HTTP/2612 and FTP/98 signature sets was more challenging, primarily because these sets contained several structurally-complex regular expressions that were difficult to determinize efficiently. For example, they contained several signatures with large counters (*i.e.,* sequences of repeating patterns) often used in combination with the alternation (*i.e.,* $re_1|re_2$) operator. Our determinizer frequently ran out of memory when attempting to construct MDFAs for such regular expressions. As an example, consider the following regular expression in HTTP/2612:

```
/.*\x2FCSuserCGI\x2Eexe\x3FLogout\x2B[^\s]{96}/i
```

[3] We were unable to compare the performance of NFA-OBDDs against DFAs because DFA construction ran out of memory. However, prior work [3] estimates that DFAs may offer throughputs of about 50 cycles/byte.

Our determinizer consumed 1.6GB of memory *for this regular expression alone*, before aborting. Producing a DFA for such regular expressions may require more sophisticated techniques, such as on-the-fly determinization [19] that are not currently implemented in our prototype. We therefore decided to exclude problematic regular expressions, and constructed MDFAs with the remaining ones (2604 for HTTP/2612 and 95 for FTP/98). Note that the MDFAs for these smaller sets of regular expressions may be *more* time-efficient and much more space-efficient than corresponding MDFAs for the entire set of regular expressions.

Figure 5 shows that in many cases NFA-OBDDs can provide throughputs comparable to those offered by MDFAs while utilizing much less memory. For example, the fastest MDFA in Figure 5(b) (constructed for a subset of 2604 signatures) offered about 50% more throughput than NFA-OBDDs, but consumed 7× more memory. The remaining MDFAs for this signature set had throughputs comparable to those of NFA-OBDDs, but consumed 270MB more memory than NFA-OBDDs. The performance gap between NFA-OBDDs and MDFAs was largest for FTP signature set, where the MDFAs (for a subset of 95 signatures) were about 4× faster than the NFA-OBDD; however, the MDFAs consumed 46MB-74MB more memory.

These results are significant for two reasons. First, conventional wisdom has long held that traditional NFAs operate *much* slower than their deterministic counterparts. This is supported by our experiments, which show that the time-efficiency of NFAs is three to four orders of magnitude slower than that of MDFAs. However, our results show that *OBDDs can drastically improve the performance of NFAs and even make them competitive with MDFAs*, which are a determinstic variant of finite automata. We believe that further enhancements to improve the time-efficiency of NFA-OBDDs can make them operate even faster than MDFAs (*e.g.,* by relaxing the OBDD data structure, and thereby eliminating several graph operations in the APPLY and RESTRICT operations).

Second, processing the set of regular expressions to produce compact yet performant MDFAs is a non-trivial exercise, often requiring time-consuming partitioning heuristics to be applied [4]. Some of the partitioning heuristics described by Yu *et al.* also require modifications to the set of regular expressions, thereby changing their semantics. Our own experience constructing MDFAs for HTTP/2612 and FTP/98 shows that this process is often challenging, especially if the regular expressions contain complex structural patterns. In contrast, NFA-OBDDs can be constructed automatically in a straightforward manner from regular expressions, including those with counters and other complex structural patterns, yet are competitive in performance and more compact than MDFAs.

Finally, we also attempted to compare the performance of NFA-OBDDs with a variant of DFAs, called hybrid finite automata (HFA) [20]. HFAs are constructed by interrupting the determinization algorithm when it encounters structurally-complex patterns (*e.g.,* large counters and $.*$ patterns) that are known to cause memory blowups when determinized. We used Becchi and Crowley's implementation [20] in our experiments, but found that it ran out of memory when trying to construct HFAs from our signature sets. For example, the HFA construction process exhausted the available memory on our machine after processing just 106 regular expressions in the HTTP/1503 set.

Deconstructing NFA-OBDD Performance. We further analyzed the performance of NFA-OBDDs to understand the time consumption of each OBDD operation. The

Operation	Fraction
AndAbstract	50%
And	39%
Map	4%
Acceptance check	7%

Fig. 6. Fraction of time spent performing OBDD operations

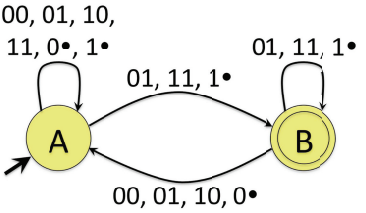


Fig. 7. 2-stride NFA for Figure 2

results reported in this section are based upon the first set of 1503 signatures; the results with the other signature sets were similar.

Figure 6 shows the fraction of time that exec_nfaobdd spends performing various OBDD operations as it processes a single input symbol. As discussed earlier, exec_nfaobdd uses the Cudd package to manipulate OBDDs. Although Cudd implements the OBDD operations described in Section 2, it also implements composite operations that combine multiple Boolean operations; the composite operations are often more efficient than performing the individual operations separately. AndAbstract is one such operation, which allows two OBDDs to be combined using an And operation followed by an existential quantification. AndAbstract takes a list of Boolean variables to be quantified, and performs the OBDD transformations needed to eliminate all these variables. The Map operation allows variables in an OBDD to be renamed, *e.g.,* it can be used to rename the $\boldsymbol{y}$ variables in $\mathcal{G}(\boldsymbol{y})$ to $\boldsymbol{x}$ variables instead.

We implemented the Boolean operations required to obtain a new frontier (described in Section 3) using one set of And, AndAbstract and Map operations. Each AndAbstract step existentially quantifies 26 Boolean variables (the $\boldsymbol{x}$ and $\boldsymbol{i}$ variables). To check whether a frontier should be accepted, we used another And operation to combine OBDD($\mathcal{F}$) and OBDD($\mathcal{A}$); the cost of an acceptance check appears in the last row of Figure 6.

Figure 6 shows that the cost of processing an input symbol is dominated by the cost of the AndAbstract and And operations to compute a new frontier. This is because the sizes of the OBDDs to be combined for frontier computation are bigger than the OBDDs that must be combined to check acceptance. Moreover, computing new frontiers involves several applications of Apply and Restrict, as opposed to an acceptance check, which requires only one Apply, thereby causing frontier computation to dominate the cost of processing an input symbol.

These results suggest that an OBDD implementation that optimizes the AndAbstract and And operations (or a relaxed variant of OBDDs that allows more efficient AndAbstract and And) can further improve the performance of NFA-OBDDs.

5 Matching Multiple Input Symbols

The preceding sections assumed that only one input alphabet is processed in each step. However, there is growing interest to develop techniques for *multi-byte matching*, *i.e.,* matching multiple input symbols in one step. Prior work has shown that multi-byte matching can improve the throughput of NFAs [21,22]. We present one such technique,

k-stride NFAs [21], and show that OBDDs can further improve the performance of k-stride NFAs.

A k-stride NFA matches k symbols of the input in a single step. Given a traditional (*i.e.,* 1-stride) ϵ-free NFA (Q, Σ, δ, q_0, F), a k-stride NFA is a 5-tuple (Q, Σ^k, Γ, q_0, F), whose input symbols are k-grams, *i.e.,* elements of Σ^k. The set of states and accepting states of the k-stride NFA are the same as those for the 1-stride NFA. Intuitively, the transition relation Γ of the k-stride NFA is computed as a k-step closure of δ, *i.e.,* $(s, \sigma_1\sigma_2 \ldots \sigma_k, t) \in \Gamma$ if and only if the state t is reachable from state s in the original NFA via transitions labeled σ_1, σ_2, ..., σ_k. The algorithm to compute Γ from δ must also consider cases where the length of the input string is not a multiple of k. Intuitively, this is achieved by padding the input string with a new "do-not-care" symbol, and introducing this symbol in the labels of selected transitions. We refer the interested reader to prior work [21,22] for a detailed description of the construction.

Figure 7 presents an example of a 2-stride NFA corresponding to the NFA in Figure 2. The do-not-care symbol is denoted by a "•". Thus, for instance, an input string 101 would be padded with • to become 101•. The 2-stride NFA processes digrams in each step. Thus, the first step would result in a transition from state A to itself A (because of the transition labeled 10), followed by a transition from A to B when it reads the second digram 1•, thereby accepting the input string.

A k-stride NFA (Q, Σ^k, Γ, q_0, F) can readily be converted into a k-stride NFA-OBDD using the same approach described in Section 3. The main difference is that the input alphabet is Σ^k (plus "•"). Transition tables of k-stride NFAs encountered in practice are generally sparse. We therefore applied a well-known technique called *alphabet compression* [21], which reduces the size of the input alphabet by combining symbols in the input alphabet into equivalence classes. An alphabet-compressed NFA can also be converted into an NFA-OBDD using the same techniques described in Section 3, and operated in the same way.

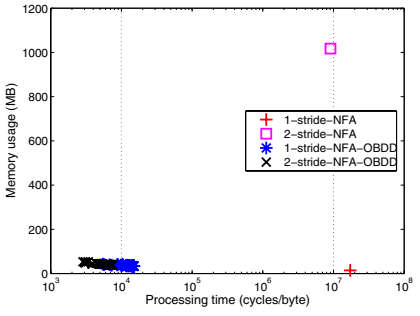
Performance of k-stride NFA-OBDDs. To evaluate the performance of k-stride NFAs and k-stride NFA-OBDDs, we used a toolchain similar to the one discussed in Section 4, but additionally applied alphabet compression. Our implementation accepts k as an input parameter. However, we have only conducted experiments for $k = 2$ because alphabet compression ran out of memory for larger values of k.

The setup that we used for the experiments reported below is identical to that described in Section 4. However, we only used two sets of Snort signatures in our measurements: (1) HTTP/1400: a subset of 1400 HTTP signatures from HTTP/1503 and (2) FTP/95 a subset of 95 FTP signatures from FTP/98. This was because the 2-stride NFA for a larger number of signatures ran out of memory during execution, thereby precluding a head-to-head comparison between the performance of 2-stride NFAs and NFA-OBDDs. We did not consider HTTP/2612 for the experiments reported in this section, because alphabet compression ran out of memory on these signature sets.

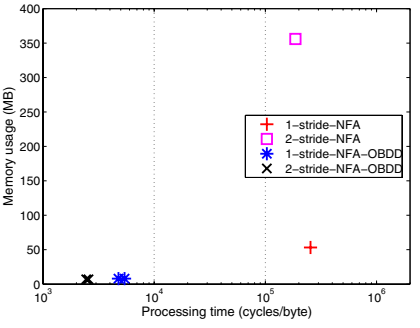
Figure 8(a) presents the size of the 1-stride and 2-stride NFA-OBDDs, and the size of the compressed alphabet. In each case, the alphabet compression algorithm took over a day to complete, and consumed over 1GB of memory. Figure 8(b) and (c) compare the performance of 1-stride NFAs and NFA-OBDDs with the performance of 2-stride

Signature Set	#States	#Transitions (1-stride)	#Transitions (2-stride)	#Alpbahet Symbols
HTTP/1400	146,992	2,246,701	44,815,280	6,928
FTP/95	15,266	3,361,065	5,136,420	848

(a) 1-stride and 2-stride NFA-OBDD construction results.



(b) HTTP/1400 regular expressions.



(c) FTP/95 regular expressions.

Fig. 8. Memory versus throughput for 1-stride NFAs, 1-stride NFA-OBDDs, 2-stride NFAs, and 2-stride NFA-OBDDs

NFAs and NFA-OBDDs. As in Section 4, for NFAs we only report the average performance across all network traces because their performance was relatively stable across all traces. We first note from Figure 8 that as expected, matching multiple bytes in the input stream improved the performance of NFAs. However, this increase in throughput comes at a drastic increase in the memory consumption of the 2-stride NFA.

In both the 1-stride and 2-stride NFAs, the use of OBDDs improved throughput—by about three orders of magnitude for HTTP/1400 and about two orders of magnitude for FTP/95. In both cases, the memory utilization of the 2-stride NFA-OBDD was smaller than that of the 2-stride NFA by two orders of magnitude. This is because OBDDs compactly encode the NFA's transition relation. These results show that *2-stride NFA-OBDDs are drastically more efficient in time and space than 2-stride NFAs.* Further investigation of the benefits of k-stride NFAs is a topic for future work.

6 Related Work

Early NIDS exclusively employed strings as attack signatures. String-based signatures are space-efficient, because their size grows linearly with the number of signatures. They are also time-efficient, and have $O(1)$ matching algorithms (*e.g.,* Aho-Corasick [23]). They are ideally suited for wire-speed intrusion detection, and have been implemented both in software and hardware [24,25,26,27,28,29]. However, prior work has shown that string-based signatures can easily be evaded by malware using polymorphism, metamorphism and other mutations [30,31,32,33]. The research community has therefore been investigating sophisticated signature schemes that require the full power of regular expressions. This in turn, has spurred both the research community to develop improved algorithms for regular expression matching, as well as NIDS vendors, who are increasingly beginning to deploy products that use regular expressions [34,35,36].

DFAs provide high-speed matching, but DFAs for large signature sets often consume gigabytes of memory. Researchers have therefore investigated techniques to improve

the space-efficiency of DFAs. These include, for example, techniques to determinize on-the-fly [19]; MDFAs, which combine signatures into multiple DFAs (as discussed in Section 4) [4]; D^2FAs [2], which reduce the memory footprint of DFAs via edge compression; and XFAs [3,8], which extend DFAs with scratch memory to store auxiliary variables, such as bitmaps and counters, and associate transitions with instructions to manipulate these variables. Some DFA variants (*e.g.,* [2,3,21]) also admit efficient hardware implementations.

These techniques use the time-efficiency of DFAs as a starting point, and seek to reduce their memory footprint. In contrast, our work uses the space-efficiency of NFAs as a starting point, and seeks to improve their time-efficiency. We believe that both approaches are orthogonal and may be synergistic. For example, it may be possible to use OBDDs to also improve the time-efficiency of MDFAs.

Our approach also provides advantages over several prior DFA-based techniques. First, it produces NFA-OBDDs from regular expressions in a fully automated way. This is in contrast to XFAs [8], which required a manual step of annotating regular expressions. Second, our approach does not modify the semantics of regular expressions, *i.e.,* the NFA-OBDDs produced using the approach described in Section 3 accept the same set of strings as the regular expressions that they were constructed from. MDFAs, in contrast, employ heuristics that relax the semantics of regular expressions to improve the space-efficiency of the resulting automata [4]. Last, because these techniques operate with DFAs, they may sometimes encounter regular expressions that are hard to determinize. For example, Smith *et al.* [8, Section 6.2] present a regular expression from the Snort data set for which the XFA construction algorithm runs out of memory. Our technique operates with NFAs and therefore does not encounter such cases.

Research on NFAs for intrusion detection has typically focused on exploiting parallelism to improve performance [9,10,37,38]. NFA operation can be parallelized in many ways. For example, a separate thread could be used to simulate each state in an NFA's frontier. Else, a set of regular expressions can be represented as a collection of NFAs, which can then be operated in parallel. FPGAs have been used to exploit this parallelism to yield high-performance NFA-based intrusion detection systems [9,10,37,38].

Although not explored in this paper, OBDDs can potentially improve NFA performance in parallel execution environments as well. For example, consider a NIDS that performs signature matching by operating a collection of NFAs in parallel. The performance of this NIDS can potentially be improved by converting it to use a collection of NFA-OBDDs instead; in this case, OBDDs improve the performance of each NFA, thereby increasing the throughput of the NIDS as a whole. Finally, NFA-OBDDs may also admit a hardware implementation. Prior work has developed techniques to implement OBDDs in CAMs [39] and FPGAs [40]. Such an implementation of NFA-OBDDs can potentially be used to improve the performance of hardware-based NFAs as well.

7 Summary

Many recent algorithms for regular expression matching have focused on improving the space-efficiency of DFAs. This paper sought to take an alternative viewpoint, and aimed to improve the time-efficiency of NFAs. To that end, we developed NFA-OBDDs,

a representation of regular expressions in which OBDDs are used to operate NFAs. Our prototype software-based implementation with Snort signatures showed that NFA-OBDDs can drastically outperform NFAs—by up to 1645× in the best case. We also showed how OBDDs can enhance the performance of NFAs that match multiple input symbols.

Acknowledgements. We thank Cristian Estan and Somesh Jha for useful discussions in the early stages of this project. We also thank Michael Bailey for shepherding the paper and the anonymous reviewers for their feedback on our work.

References

1. Becchi, M.: Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation. PhD thesis, Washington University in St. Louis (2009)
2. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: ACM SIGCOMM Conference, pp. 339–350. ACM, New York (2006)
3. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the Big Bang: Fast and scalable deep packet inspection with extended finite automata. In: SIGCOMM Conference, pp. 207–218. ACM, New York (2008)
4. Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and memory-efficient regular expression matching for deep packet inspection. In: ACM/IEEE Symp. on Arch. for Networking and Comm. Systems, pp. 93–102 (2006)
5. Roesch, M.: Snort - lightweight intrusion detection for networks. In: USENIX Conf. on System Administration, USENIX, pp. 229–238 (1999)
6. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley, Reading (2007)
7. Becchi, M., Crowley, P.: An improved algorithm to accelerate regular expression evaluation. In: Intl. Conf. on Architectures for Networking and Communication Systems, pp. 145–154. ACM, New York (2007)
8. Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: Symp. on Security and Privacy, pp. 187–201. IEEE Computer Society, Los Alamitos (2008)
9. Sidhu, R., Prasanna, V.: Fast regular expression matching using FPGAs. In: Symp. on Field-Programmable Custom Computing Machines, pp. 227–238. IEEE Computer Society, Los Alamitos (2001)
10. Clark, C.R., Schimmel, D.E.: Scalable pattern matching for high-speed networks. In: IEEE Symp. on Field-Programmable Custom Computing Machines, pp. 249–257. IEEE Computer Society, Los Alamitos (2004)
11. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
12. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, J.: Symbolic model checking: 10^{20} states and beyond. In: Symp. on Logic in Computer Science, pp. 401–424. IEEE Computer Society, Los Alamitos (1990)
13. Thompson, K.: Programming techniques: Regular expression search algorithm. ACM Commun. 11(6), 419–422 (1968)
14. Somenzi, F.: CUDD: CU decision diagram package, release 2.4.2 Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder, `http://vlsi.colorado.edu/~fabio/CUDD`

15. Signatures referenced in Section 4 and Section 5, http://www.cs.rutgers.edu/~vinodg/papers/raid2010
16. PCRE: The Perl compatible regular expression library, http://www.pcre.org
17. Cox, R.: Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...) (2007), http://swtch.com/$\sim$rsc/regexp/regexp1.html.
18. Smith, R., Estan, C., Jha, S.: Backtracking algorithmic complexity attacks against a NIDS. In: Annual Computer Security Applications Conf., pp. 89–98. IEEE Computer Society, Los Alamitos (2006)
19. Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: Conf. on Computer and Comm. Security, pp. 262–271. ACM, New York (2003)
20. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: Intl. Conf. on emerging Networking EXperiments and Technologies (2007)
21. Brodie, B.C., Taylor, D.E., Cytron, R.K.: A scalable architecture for high-throughput regular-expression pattern matching. In: Intl. Symp. Computer Architecture, pp. 191–202. IEEE Computer Society, Los Alamitos (2006)
22. Becchi, M., Crowley, P.: Efficient regular expression evaluation: Theory to practice. In: Intl. Conf. on Architectures for Networking and Communication Systems, pp. 50–59. ACM, New York (2008)
23. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. ACM Comm. 18(6), 333–340 (1975)
24. Dharmapurikar, S., Lockwood, J.W.: Fast and scalable pattern matching for network intrusion detection systems. Jour. on Selected Areas in Comm. 24(10), 1781–1792 (2006)
25. Liu, R., Huang, N., Chen, C., Kao, C.: A fast string-matching algorithm for network processor-based intrusion detection system. Trans. on Embedded Computing Sys. 3(3), 614–633 (2004)
26. Sourdis, I., Pnevmatikatos, D.: Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In: Cheung, P., Constantinides, G., Sousa, J. (eds.) FPL 2003. LNCS, vol. 2778, pp. 880–889. Springer, Heidelberg (2003)
27. Tan, L., Sherwood, T.: A high throughput string matching architecture for intrusion detection and prevention. In: Intl. Symp. Computer Architecture, pp. 112–122. IEEE Computer Society, Los Alamitos (2005)
28. Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic memory-efficient string matching algorithms for intrusion detection. In: IEEE INFOCOM 2004, pp. 333–340. IEEE Computer Society, Los Alamitos (2004)
29. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High performance network intrusion detection using graphics processors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 116–134. Springer, Heidelberg (2008)
30. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: Usenix Security, p. 9. USENIX (2001)
31. Jordan, M.: Dealing with metamorphism. Virus Bulletin Weekly (2002)
32. Ptacek, T., Newsham, T.: Insertion, evasion and denial of service: Eluding network intrusion detection, http://insecure.org/stf/secnet_ids/secnet_ids.html.
33. Shankar, U., Paxson, V.: Active mapping: Resisting NIDS evasion without altering traffic. In: Symp. on Security and Privacy, pp. 44–61. IEEE Computer Society, Los Alamitos (2003)
34. TippingPoint, http://www.tippingpoint.com
35. LSI-Corporation: Tarari RegEx content processor, http://www.tarari.com
36. Cisco: IOS terminal services configuration guide, http://tinyurl.com/2eouvq

37. Hutchings, B.L., Franklin, R., Carver, D.: Assisting network intrusion detection with reconfigurable hardware. In: Annual Symp. on Field-Programmable Custom Computing Machines, pp. 111–120. IEEE Computer Society, Los Alamitos (2002)
38. Mitra, A., Najjar, W., Bhuyan, L.: Compiling PCRE to FPGA for accelerating Snort IDS. In: Symp. on Arch. for Networking and Comm. Systems, pp. 127–136. ACM, New York (2007)
39. Yusuf, S., Luk, W.: Bitwise optimized CAM for network intrusion detection systems. In: Intl. Conf. on Field Prog. Logic and Applications, pp. 444–449. IEEE Press, Los Alamitos (2005)
40. Sinnappan, R., Hazelhurst, S.: A reconfigurable approach to packet filtering. In: Brebner, G., Woods, R. (eds.) FPL 2001. LNCS, vol. 2147, pp. 638–642. Springer, Heidelberg (2001)

GrAVity: A Massively Parallel Antivirus Engine

Giorgos Vasiliadis and Sotiris Ioannidis

Institute of Computer Science, Foundation for Research and Technology – Hellas,
N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece
{gvasil,sotiris}@ics.forth.gr

Abstract. In the ongoing arms race against malware, antivirus software is at the forefront, as one of the most important defense tools in our arsenal. Antivirus software is flexible enough to be deployed from regular users desktops, to corporate e-mail proxies and file servers. Unfortunately, the signatures necessary to detect incoming malware number in the tens of thousands. To make matters worse, antivirus signatures are a lot longer than signatures in network intrusion detection systems. This leads to extremely high computation costs necessary to perform matching of suspicious data against those signatures.

In this paper, we present GrAVity, a massively parallel antivirus engine. Our engine utilized the compute power of modern graphics processors, that contain hundreds of hardware microprocessors. We have modified ClamAV, the most popular open source antivirus software, to utilize our engine. Our prototype implementation has achieved end-to-end throughput in the order of 20 Gbits/s, 100 times the performance of the CPU-only ClamAV, while almost completely offloading the CPU, leaving it free to complete other tasks. Our micro-benchmarks have measured our engine to be able to sustain throughput in the order of 40 Gbits/s. The results suggest that modern graphics cards can be used effectively to perform heavy-duty anti-malware operations at speeds that cannot be matched by traditional CPU based techniques.

1 Introduction

The ever increasing amount of malicious software in todays connected world, poses a tremendous challenge to network operators, IT administrators, as well as ordinary home users. Antivirus software is one of the most widely used tools for detecting and stopping malicious or unwanted software. For an effective defense, one needs virus-scanning performed at central network traffic ingress points, as well as at end-host computers. As such, anti-malware software applications scan traffic at e-mail gateways and corporate gateway proxies, and also on edge compute devices such as file servers, desktops and laptops. Unfortunately, the constant increase in link speeds, storage capacity, number of end-devices and the sheer number of malware, poses significant challenges to virus scanning applications, which end up requiring multi-gigabit scanning throughput.

Typically, a malware scanner spend the bulk of its time matching data streams against a large set of known *signatures*, using a pattern matching algorithm.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 79–96, 2010.

Pattern matching algorithms analyze the data stream and compare it against a database of signatures to detect known malware. The signature patterns can be fairly complex, composed of different-size strings, wild-card characters, range constraints, and sometimes recursive forms. Every year, as the amount of malware grows, the number of signatures is increasing proportional, exposing scaling problems of anti-malware products.

To come up with the large signature sets, most approaches rely on the quickly, fast and accurate filtering of the "no-match" cases, based on the fact that the majority of network traffic and files is not supposed to contain viruses [9]. Other approaches are based on specialized hardware, like FPGAs and ASICs, to achieve high performance [15,14]. Such hardware solutions are very efficient and perform quite well, however they are hard to program, complex to modify, and are usually tied to a specific implementation.

In contrast, commodity graphics processing units (GPUs) have been proven to be very efficient and highly effective at accelerating the pattern matching operations of network intrusion detection systems (NIDS) [26,21,27]. Driven by the ever-growing video game industry, modern GPUs have been constantly evolving to ever more powerful and flexible stream processors, specialized for computationally-intensive and highly parallel operations. The massive number of transistors devoted to data processing, rather than data caching and flow control, can be exploited to perform computations that up till now were handled by the CPU.

In this work, we explore how the highly parallel capabilities of commodity graphics processing units can be utilized to improve the performance of malware scanning programs and how they can assist and offload the CPU whenever possible.

From a high-level view, malware scanning is divided into two phases. First, all files are scanned by the GPU, using a combined DFA state machine that contain only a prefix from each signature. This results in identifying all potentially malicious files, but a number of clean files as well. The GPU then outputs a set of suspect matched files and the corresponding offsets in those files. In the second phase, all those files are rescanned using a full pattern matching algorithm.

The contributions of our work are:

- We have designed, implemented and evaluated a pattern matching algorithm on modern GPUs. Our implementation could be adapted to any other multi-core system, as well.
- We integrated our GPU implementation into ClamAV [12], the most popular and widely used open-source virus scanning software, proving that our solution can be used in the real-world.
- We developed and implemented a series of system level optimizations to improve end-to-end performance of our system.
- We implemented, experimented and analyzed our GPU-assisted virus scanning application with various configurations and we show that modern GPUs can effectively be used, in coordination with the CPU, to drastically improve the performance of anti-malware applications.

Our prototype implementation, called GrAVity, achieved a scanning throughput of 20 Gbits/s for binary files. This represents a speed-up factor of 100 from the single CPU-core case. Also, in special cases, where data is cached on the graphics card, the scanning throughput can reach 110 Gbits/s.

The rest of the paper is organized as follows. In Section 2, we present some background on general-purpose GPU (GPGPU) programming and introduce the related virus scanning architectures. The architecture and acceleration techniques are presented in Section 3. The performance analysis and evaluation are given in Section 4. The paper ends with an outline of related work in Section 5 and some concluding remarks in Section 6.

2 Background

In this section, we briefly describe the architecture of modern graphics cards and the general-purpose computing functionality they provide for non-graphics applications. We also discuss some general aspects of virus-scanning techniques.

2.1 GPU Programming

For our work we selected the NVIDIA GeForce 200 Series architecture, which offers a rich programming environment and flexible abstraction models through the Compute Unified Device Architecture (CUDA) SDK [18]. The CUDA programming model extends the C programming language with directives and libraries that abstract the underlying GPU architecture and make it more suitable for general purpose computing. In contrast with standard graphics APIs, such as OpenGL and DirectX, CUDA exposes several hardware features to the programmer. The most important of these features is the existence of convenient data types, and the ability to access the DRAM of the device card through the general memory addressing mode it provides. CUDA also offers highly optimized data transfer operations to and from the GPU.

The GeForce 200 Series architecture, in accordance with its ancestors GeForce 8 (G80) and GeForce 9 (G90) Series, is based on a set of multiprocessors, each of which contains a set of *stream processors* operating on SIMD (Single Instruction Multiple Data) programs. When programmed through CUDA, the GPU can be used as a general purpose processor, capable of executing a very high number of threads in parallel.

A unit of work issued by the host computer to the GPU is called a *kernel*, and is executed on the device as many different *threads* organized in *thread blocks*. Each multiprocessor executes one or more thread blocks, with each group organized into *warps*. A warp is a fraction of an *active group*, which is processed by one multiprocessor in one batch. Each of these warps contains the same number of threads, called the *warp size*, and is executed by the multiprocessor in a SIMD fashion. Active warps are time-sliced: A thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessors computational resources.

Stream processors within a processor share an instruction unit. Any control flow instruction that causes threads of the same warp to follow different execution paths reduces the instruction throughput, because different executions paths have to be serialized. When all the different execution paths have reached a common end, the threads converge back to the same execution path.

A fast *shared memory* is managed explicitly by the programmer among thread blocks. The *global*, *constant*, and *texture memory spaces* can be read from or written to by the host, are persistent across kernel launches by the same application, and are optimized for different memory usages [18]. The constant and texture memory accesses are cached, so a read from them costs much less compared to device memory reads, which are not being cached. The texture memory space is implemented as a read-only region of device memory.

2.2 Virus Scanning and ClamAV

ClamAV [12] is the most widely used open-source virus scanner. It offers client-side protection for personal computers, as well as mail and file servers used by large organizations. As of January 2010, it has a database of over 60,000 virus signatures, and consists of a core scanner library and various command-line utilities. The database includes signatures for non-polymorphic viruses in simple string format, and for polymorphic viruses in regular expression format (polymorphic signatures).

The current version of ClamAV uses an optimized version of the Boyer-Moore algorithm [3] to detect non-polymorphic viruses using simple fixed string signatures. For polymorphic viruses, on the other hand, ClamAV uses a variant of the classical Aho-Corasick algorithm [1].

The Boyer-Moore implementation in ClamAV, uses a *shift-table* to reduce the number of times the Boyer-Moore routine is called. At start up, ClamAV preprocess every signature and stores the shift value of every possible block (arbitrarily choosing a block size of 3 bytes) to initialize a shift table. Then, at any point in the input stream, ClamAV can determine if it can skip up to three bytes by performing a quick hash on them. ClamAV also creates a hash table based on the first three bytes of the signature and uses this table at run-time when the shift table returns a match. Since this algorithm uses hash functions on all bytes of a signature, it is only usable against non-polymorphic viruses.

The Aho-Corasick implementation uses a trie to store the automaton generated from the polymorphic signatures. The fixed string parts of each polymorphic signatures are extracted, and are used to build a trie. At the scanning phase, the trie will be used to scan for all these fixed parts of each signature simultaneously. For example, the signature `‘‘495243*56697275’’` contains two parts, `‘‘495243’’` and `‘‘56697275’’`, which are matched individually by the Aho-Corasick algorithm. When all parts of a signature are found, ClamAV also verifies the order and the gap between the parts, as specified in the signature. To quickly perform a lookup in this trie, ClamAV uses a 256 element array for each node. In the general case, the trie has a variable height, and all patterns beginning with the same prefix are stored under the corresponding leaf node.

However, in order to simplify the trie construction, the height is restricted to be equal to the size of the shortest part in the polymorphic signatures, which is currently equal to two. Thus, the trie depth is fixed to two and all patterns are stored at the same trie level. During the scanning phase, ClamAV scans an input file and detects occurrences of each of the polymorphic signatures, including partially and completely overlapping occurrences. The Aho-Corasick algorithm has the desirable property that the processing time does not depend on the size or number of patterns in a significant way.

The main reason that ClamAV uses both Boyer-Moore and Aho-Corasick is that many parts in the polymorphic signatures are short, and they restrict the maximum shift distance allowed (bounded by the shortest pattern) in the Boyer-Moore algorithm. Matching the polymorphic signatures in Aho-Corasick avoid this problem. Furthermore, compared with the sparse automaton representation of the Aho-Corasick algorithm, the compressed shift table is a more compact representation of a large number of non-polymorphic signatures in fixed strings, so the Boyer-Moore algorithm is more efficient in terms of memory space.

3 Design and Implementation

GrAVity utilizes the GPU to quickly filter out the data segments that do not contain any viruses. To achieve this, we have modified ClamAV, such that the input data stream is initially scanned by the GPU. The GPU uses a prefix of each virus signature to quickly filter-out clean data. Most data do not contain any viruses, so such filtering is quite efficient as we will see in Section 4.

The overall architecture of GrAVity is shown in Figure 1. The contents of each file are stored into a buffer in a region of main memory that can be transferred via DMA into the memory of the GPU. The SPMD operation of the GPU is ideal

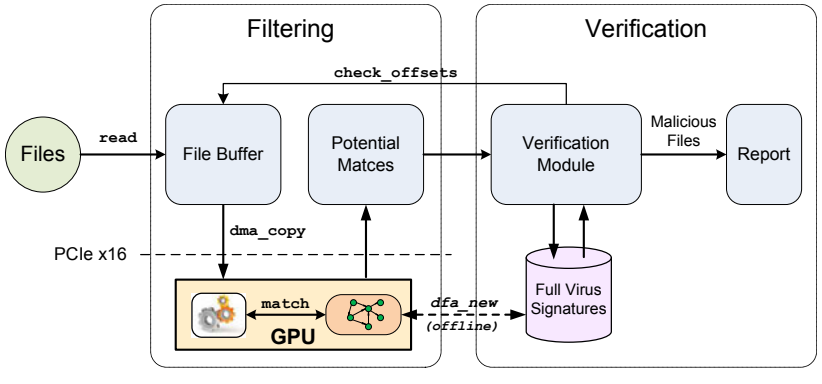


Fig. 1. GrAVity Architecture. Files are mapped onto pinned memory that can be copied via DMA onto the graphics card. The matching engine performs a first-pass filtering on the GPU and return potential true positives for further checking onto the CPU.

for creating multiple search engine instances that will scan for virus signatures on different data in a massively parallel fashion. If the GPU detects a suspicious virus, that is, there is prefix match, the file is passed to the verification module for further investigation. If the data stream is clean, no further computation takes place. Therefore, the GPU is employed as a first-pass high-speed filter, before completing any further potential signature-matching work on the CPU.

3.1 Basic Mechanisms

At start-up, the entire signature set of ClamAV is preprocessed, to construct a deterministic finite automaton (DFA). Signature matching using a DFA machine has linear complexity as a function of the input text stream, which is very efficient. Unfortunately, the number of virus signatures, as well as their individual size is quire very large, so it may not be always feasible to construct a DFA machine that will contain the complete signature set. As the number and size of matching signatures increase, the size of the automaton also increases.

To overcome this, we chose to only use a portion from each virus signature. By using the first n symbols from each signature, the height of the corresponding DFA matching machine is limited to n, as shown in Figure 2. In addition, all patterns that begin with the same prefix are stored under the same node, called *final node*. In case the length of the signature pattern is smaller than the prefix

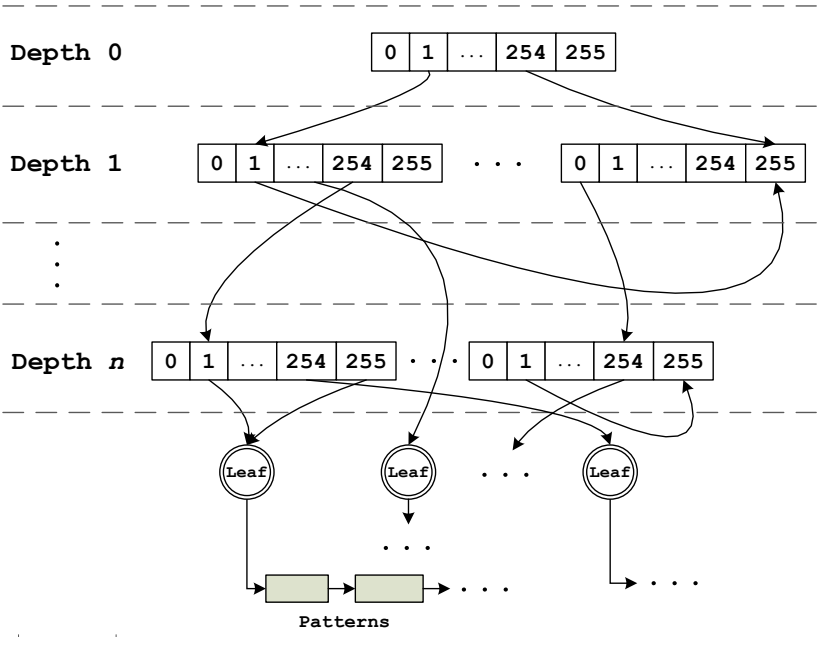


Fig. 2. A fragment of the DFA structure with n levels. The patterns beginning with the same prefix are stored under the same final node (leaf).

length, the entire pattern is added. A prefix may also contain special characters, such as the wild-characters * and ?, that are used in ClamAV signatures to describe a known virus.

At the scanning phase, the input data will be initially scanned by the DFA running on the GPU. Obviously, the DFA may not be able to match an exact virus signature inside a data stream, as in many cases the length of the signature is longer than the length of the prefix we used to create the automaton. This will be the first-level filtering though, designed to offload the bulk of the work from the CPU, by drastically eliminating a significant portion of the input data that need to be scanned.

It is clear that the longer the prefix, the fewer the number of false positives at this initial scanning phase. As we will see in Section 4, using a value of 8 for n, can result to less than 0.0001% of false positives in a realistic corpus of binary files.

3.2 Parallelizing DFA Matching on the GPU

During scan time, the algorithm moves over the input data stream one byte at a time. For each byte, the scanning algorithm moves the current state appropriately. The pattern matching is performed byte-wise, meaning that we have an input width of 8 bits and an alphabet size of $2^8 = 256$. Thus, each state will contain 256 pointers to other states, as shown in Figure 2. The size of the DFA state machine is thus $|\#States| * 1024$ bytes, where every pointer occupies 4 bytes of storage.

If a final-state is reached, a potential signature match has been found. Consequently, the offset where the match has been found is marked and all marked offsets will be verified later by the CPU. The idea is to quickly weed-out the dominant number of true negatives using the superior performance and high parallelism of the GPU, and pass on the remaining potential true positives to the CPU.

To utilize all streaming processors of the GPU, we exploit its data parallel capabilities by creating multiple threads. An important design decision is how to assign the input data to each thread. The simplest approach would be to use multiple data input streams, one for each thread, in separate memory areas. However, this will result in asymmetrical processing effort for each processor and will not scale well. For example, if the sizes of the input streams vary, the amount of work per thread will not be the same. This means that threads will have to wait, until all have finished searching the data stream that was assigned to them.

Therefore, each thread searches a different portion of the input data stream, at the matching phase. To best utilize the data-parallel capabilities of the GPU, we create a large number of threads that run simultaneously. Our strategy splits the input stream in distinct chunks, and each chunk is processed by a different thread. Figure 3 shows how each GPU thread scans its assigned chunk, using the underlying DFA state table. Although they access the same automaton, each thread maintains its own state, eliminating any need for communication between them.

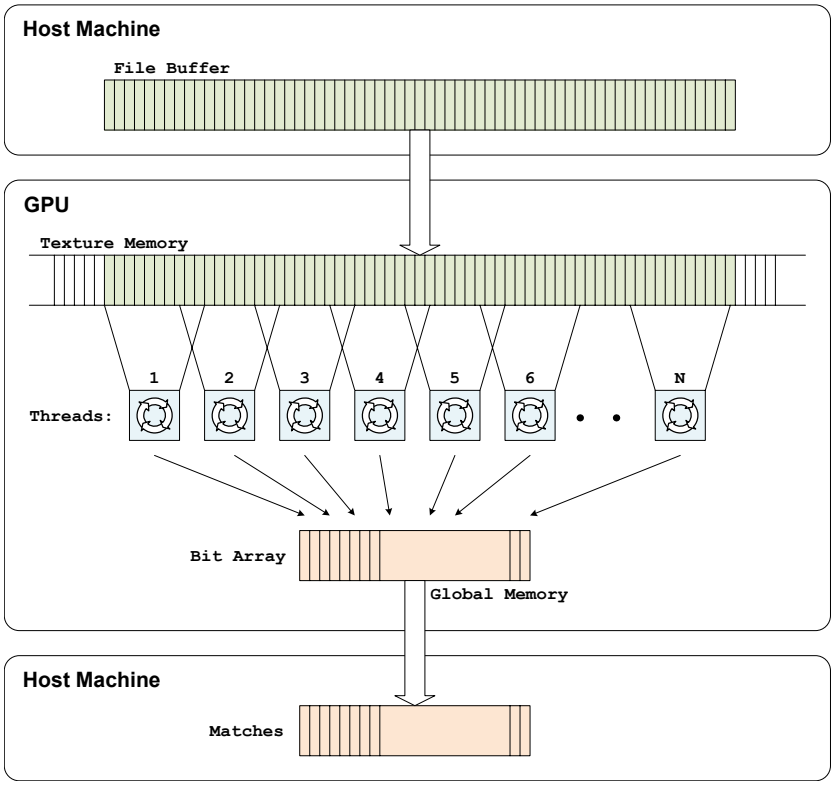


Fig. 3. Pattern matching on the GPU

A special case, however, is for patterns that may span across two or more different chunks. The simplest approach for fixed string patterns, would be to process in addition, n bytes, where n is the maximum pattern length in the dictionary. Unfortunately, the virus patterns are usually very large, as shown in Figure 4 for the ClamAV, especially when compared with patterns in other pattern matching systems like Snort. Moreover, a regular expression may contain the wild card character $*$, thus the length of the patterns may not be determined. To solve this problem, we used the following heuristic: each thread continues the search up to the following chunk (which contains the consecutive bytes), until a fail or final-state is reached. While matching a pattern that spans chunk boundaries, the state machine will perform regular transitions. However, if the state machine reaches a fail or final-state, then it is obvious that there is no need to process the data any further, since any consecutive patterns will be matched by the thread that was assigned to search the current chunk. This allows the threads to operate independently and avoid any communication between them, regarding boundaries in the input data buffer.

Every time a match is found, it is stored to a bit array. The size of the bit array is equal to the size of the data that is processed at concurrently. Each bit in

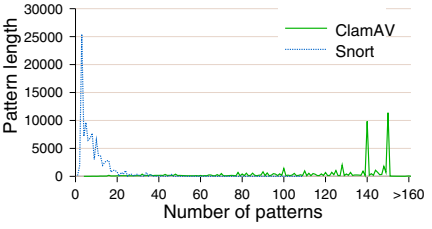


Fig. 4. ClamAV pattern length distribution

the array represents whether a match was found in the corresponding position. We have chosen the bit array structure, since it is a compact representation of the results, even in the worst case scenario where a match is found at every position.

3.3 Optimized Memory Management

The two major tasks of DFA matching, is determining the address of the next state in the state table, and fetching the next state from the device memory. These memory transfers can take up to several hundreds of nanoseconds, depending on the traffic conditions and congestion.

Our approach for hiding memory latencies is to run many threads in parallel. Multiple threads can improve the utilization of the memory subsystem, by overlapping data transfer with computation. To obtain the highest level of performance, we tested GrAVity to determine how the computational throughput is affected by the number of threads. As discussed in Section 4.2 the memory subsystem is best utilized when there is a large number of threads, running in parallel.

Moreover, we have investigated storing the DFA state table both in the global memory space, as well as in the texture memory space of the graphics card. The texture memory can be accessed in a random fashion for reading, in contrast to global memory, where the access patterns must be coalesced. This feature can be very useful for algorithms like DFA matching, which exhibit irregular access patterns across large data sets. Furthermore, texture fetches are cached, increasing the performance when read operations preserve locality. As we will see in Section 4.2, the usage of texture memory can boost the computational throughput up to a factor of two.

3.4 Other Optimizations

In addition to optimizing the memory usage, we considered two other optimizations: the use of page-locked (or pinned) memory, and reducing the number of transactions between the host and the GPU device.

The page-locked memory offers better performance, as it does not get swapped (i.e. non-pageable memory). Furthermore, it can be accessed directly by the GPU

through Direct Memory Access (DMA). Hence, the usage of page-locked memory improves the overall performance, by reducing the data transferring costs to and from the GPU. The contents of the files are read into a buffer allocated from page-locked memory, through the CUDA driver. The DMA then, transfers the buffer from the physical memory of the host, to the texture memory of the GPU.

To further improve performance, we use a large buffer to store the contents of many files, that is transferred to the GPU in a single transaction. The motivation behind this feature, is that the matching results will be the same, whether we scan each file individually or scanning several files back-to-back, all at once. This results in a reduction of I/O transactions over the PCI Express bus.

4 Performance Evaluation

In this section, we evaluate our prototype implementation. First, we give a short description of our experimental setup. We then present an overall performance comparison of GrAVity and ClamAV, as well as detailed measurements to show how it scales with the prefix length and the number of threads that are executing on the GPU.

4.1 Experimental Environment

For our experiment testbed, we used the NVIDIA GeForce GTX295 graphics card. The card consists of two PCBs (Printed Circuit Board), each of which is equipped with 240 cores, organized in 30 multiprocessors, and 896MB of GDDR3 memory. Our base system is equipped with two Intel(R) Xeon(R) E5520 Quad-core CPUs at 2.27GHz with 8192KB of L2-cache, and a total of 12GB of memory. The GPU is interconnected using a PCIe 2.0 x16 bus.

We use the latest signatures set of ClamAV (main v.52, released on February 2010). The set consists of 60 thousand string and regular expression signatures. As input data stream, we used the files under `/usr/bin/` in a typical Linux installation. The directory contains 1516 binary files, totalling about 132MB of data. The files do not contain any virus, however they exercise most code branches of GrAVity.

In all experiments we conducted, we disregarded the time spent in the initialization phase for both ClamAV and GrAVity. The initialization phase includes the loading of the patterns and the building of the internal data structures, so there is no actual need to include this time in our graphs.

4.2 Microbenchmarks

Figure 5 shows the matching throughput for varying signature prefix lengths. We explore the performance that different types of memory can provide, by using global device and texture memory respectively to store the DFA state table. The horizontal axis shows the signature prefix length. We also repeated the experiment using different number of threads. As the number of threads

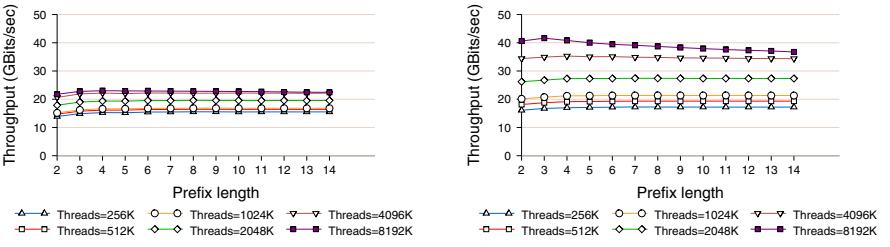


Fig. 5. Sustained throughput for varying signature prefix. Higher number of threads achieve higher performance as memory latencies are hidden. We demonstrate the effect of different GPU memory types on performance. (a) uses global device memory to store the DFA state table, where (b) uses texture memory.

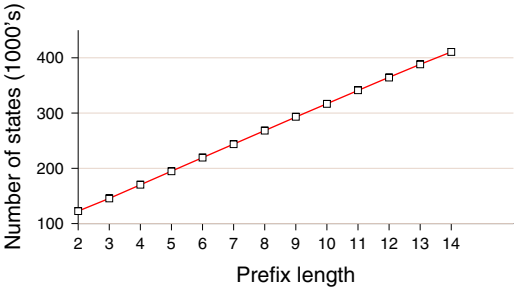


Fig. 6. Memory requirements for the storage of the DFA as a function of the signature prefix length

increases, the throughput sustained by the GPU also increases. When using eight millions threads, which is the maximum acceptable number of threads for our application, the computational throughput raises to a maximum of 40 Gbits/s.

Comparing the two types of memory available in the graphics card, we observe that the texture memory significantly improves the overall performance by a factor of two. The irregularity of memory accesses that DFA matching exhibits, can be partially hidden when using texture memory. Texture memory provides a random access model for fetching data, in contrast with global memory where access patterns have to be coalesced. Moreover, texture fetches are cached, which offers an additional benefit.

The total memory requirements for storing the DFA, independently of the memory type, is shown in Figure 6. We observe that the total number of states of the DFA machine is growing linearly to the length of the prefix. Using a value of 14 as a prefix length, results in a DFA machine that holds about 400 thousands states. In our DFA implementation this is approximately 400MB of memory — each state requires 1KB of memory.

4.3 Application Performance

In this section, we evaluate the overall performance of GrAVity. Each experiment was repeated a number of times, to ensure that all files were cached by the operating system. Thus, no file data blocks were read from disk during our experiments. We have verified the absence of I/O latencies using the `iostat(1)` tool.

Throughput. In this experiment we evaluate the performance of GrAVity compared to vanilla ClamAV. Figure 7 shows the throughput achieved for different prefix lengths. The overall throughput increases rapidly, raising at a maximum of 20 Gbits/s. A plateau is reached for a prefix length of around 10.

As the prefix length increases, the number of potential matches decreases, as shown in Figure 9. This results to lower CPU post-processing, hence the overall application throughput increases. In the next section, we investigate in more detail the breakdown of the execution time.

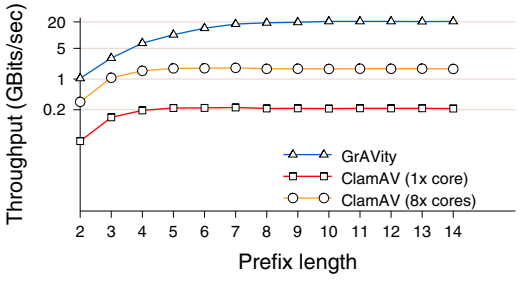


Fig. 7. Performance of GrAVity and ClamAV. We also include the performance number for ClamAV running on 8 cores. The CPU-only performance is still an order of magnitude less that the GPU-assisted. The numbers demonstrate that additional CPU cores offer less benefit than that of utilizing the GPU.

Execution Time Breakdown. We measure the execution time for data transfers, result transfers, CPU and GPU execution. We accomplish this by adding performance counters before each task.

As expected, Figure 8 shows that for small prefix sizes most of the time is dominated by the cost of the CPU, verifying the possible matches reported back by the GPU. For example, for a prefix length equal to 2, approximately 95% of the total execution time is spent on the CPU to validate the potential matches. For a prefix length equal to 14, the corresponding CPU time results in just 20% of the total execution time, and in actual time signifies a reduction of *three orders of magnitude*, while the GPU consumes 54% of the total execution. As the prefix length increases, this overhead decreases and the GPU execution time becomes the dominant factor. For verification, in Figure 9 we plot the number of potential matches reported in accordance with the signature prefix length.

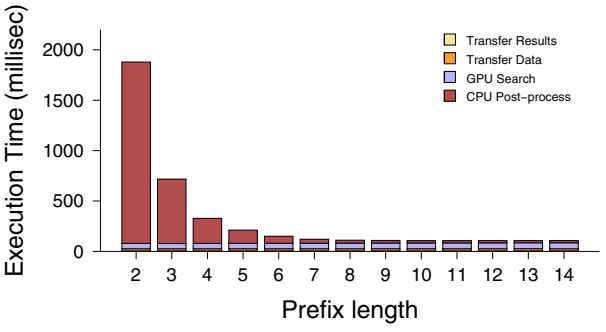


Fig. 8. GrAVity execution time breakdown

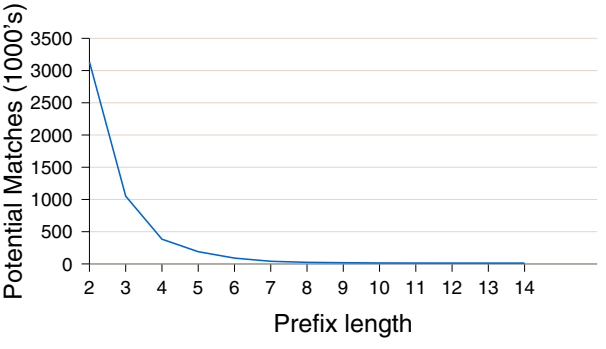


Fig. 9. Number of matches as a function of the signature prefix length

4.4 Scaling Factor

To measure how our GPU pattern matching implementation has improved during the evolution of GPU models, we used three additional older-generation graphics cards. Specifically, we utilized a GeForce 8600GT, which was released early on March 2007, a GeForce8800GT released on December 2007, and a GeForce 9800GX2 that released 4 months later, on March 2008.

Figure 10 shows that in less than two years, the computational throughput has raised 20 times, from about 2 Gbits/sec to over 40 Gbits/sec. For comparison reasons, we also calculated and included the respective numbers of various generations of CPUs.

4.5 Peak Performance

In the final experiment we explore the ideal performance our GPU implementation can achieve. For this reason, we created a large file containing the `NULL` character, to ensure that no state transitions will be performed at the matching phase. The automaton will remain always at the same state, which will be cached. Moreover, no matches will be reported, that would trigger an expensive memory write at the global device memory. In this "best-case" scenario,

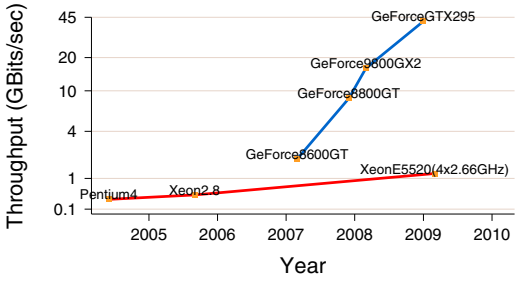


Fig. 10. Performance sustained by our pattern matching implementation on different generation of GPU and CPU models

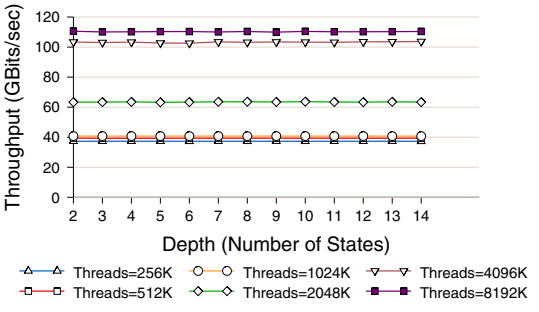


Fig. 11. Peak performance sustained by our pattern matching implementation on the GPU

our throughput reached an order of 110 Gbits/s. This demonstrated the top end performance the hardware can support. GrAVity's end-to-end performance reaches a very respectable 20% of this upper bound.

5 Related Work

Multi-pattern matching algorithms is one of the core operations used by applications in many domains. In the networking area, the most important applications, that primarily rely on pattern matching, are intrusion detection systems and malware scanners.

Many approaches rely on the hardware implementation of pattern matching algorithms, like FPGAs [20,22,10,2], CAMs [24,29,23] and Network Processors [6,7]. Most of these studies have focused primarily on network intrusion detection systems, which are quite different from virus scanning applications [9].

Recently, however, several efforts have been made to improve the performance of ClamAV [16,9,15,14]. Many approaches rely on the simple, fast and accurate filtering of the input data stream, as software implementations running on generic processors [16,9], or more complex approaches using specialized hardware [15,14].

Recent software implementations have adapted Bloom filters for use in virus scanning as a first-level filter before the exact pattern matching algorithm occurs [9,5]. A fragment of constant length is extracted from every signature and inserted into a Bloom filter. At the scanning phase, a window of the same size slides over the files to be examined, and its content at every position is tested against the filter. A Bloom filter is the most compact structure that can store a dictionary and is used to determine whether a string belongs to that dictionary or not. A major drawback of Bloom filters, however, is that they cannot be used for regular expressions matching. A possible solution is to select an invariant fragment (i.e. a fixed byte sequence) from a wild-card containing signature and put it in the filter. Unfortunately, the fact that the fragments have to be of the same length, will shorten the hashing window to the shortest signature or fragment, and will increase the false positive rate. Several approaches have been used Bloom filters efficiently in specialized hardware, for example with FPGAs [8,17,4]. Hardware implementations provide better performance, although with a high, and often prohibitive, cost for many organizations.

Besides specialized hardware solutions, commodity multi-core processors have begun gaining popularity, primarily due to their increased computing power and low cost. It has been shown that fixed-string pattern matching implementations on SPMD processors, such as the IBM Cell processor, can achieve a computational throughput of up to 2.2 Gbits/s [19], while regular expression matching up to 7.5 Gbits/s [13]. In the context of network intrusion detection systems, graphics processors have been used to accelerate their performance [26,27,21,11,28,25]. Specifically, work in [26,27] significantly improved the performance of Snort by offloading the string searching and regular expression matching operations to the GPU. The work in this paper, exploits and extends some of those ideas and applies them in a hybrid, GPU-CPU malware detection architecture, with a drastic improvement in performance.

6 Conclusions

In this paper, we presented GrAVity, a massively parallel antivirus engine that utilizes the GPU to offload the bulk of pattern and regular expression matching from a popular antivirus system. Our system exploits the highly threaded architecture of modern graphics processors, as well as the embarrassingly parallel nature of virus scanning to achieve end-to-end throughput in the order of 20 Gbits/s. This result is 100 times faster than the unmodified ClamAV running on a modern CPU. Our benchmarks also showed that our approach completely offloads the CPU and frees it to perform other tasks. Finally, our micro-benchmarks showed that it is possible to achieve throughput in the order of 40 Gbits/s in cases where data is pre-cached on the graphics card, showing that solving data transfer bottlenecks can lead to doubling of performance.

To achieve such high performance, we tuned our system and performed a number of optimizations. Since virus signatures are both very long and more numerous compared to other signature matching systems, like network intrusion

detection systems, we build our engine as a pre-filter, that uses prefixes of the actual signatures. These prefixes are used to create the DFAs used in the actual pattern matching on the GPU. Our architecture also takes advantage of the physical memory hierarchies of graphics processors, as well as, bulk data transfers using DMA.

As future work we plan to investigate how to port our engine to commercial antivirus software, as well, other tools such as antispyware. In terms of architecture, we plan to overlap GPU and CPU matching phase, as right now our system is serialized in that respect. Finally we plan on utilizing multiple GPUs instead of a single one. Modern motherboards, such as the one we used in our evaluation, support multiple GPUs on the PCI Express bus. In our case it would be possible to utilize up to four such cards. Such a system would require a more thorough investigation of communication and synchronization between multiple GPUs.

Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS. Giorgos Vasiliadis and Sotiris Ioannidis are also with the University of Crete.

References

1. Aho, A.V., Corasick, M.J.: Efficient String Matching: an Aid to Bibliographic Search. Communications of the ACM 18(6), 333–340 (1975)
2. Baker, Z.K., Prasanna, V.K.: Time and area efficient pattern matching on FPGAs. In: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA 2004), pp. 223–232. ACM, New York (2004)
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the Association for Computing Machinery 20(10), 762–772 (1977)
4. Braun, F., Lockwood, J., Waldvogel, M.: Protocol wrappers for layered network packet processing in reconfigurable hardware. IEEE Micro 22(1), 66–74 (2002)
5. Cha, S.K., Moraru, I., Jang, J., Truelove, J., Brumley, D., Andersen, D.G.: SplitScreen: Enabling efficient, distributed malware detection. In: Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Jose, CA (April 2010)
6. Clark, C.R., Lee, W., Schimmel, D.E., Contis, D., Kon, M., Thomas, A.: A Hardware Platform for Network Intrusion Detection and Prevention. In: Crowley, P., Franklin, M.A., Hadimioglu, H., Onufryk, P.Z. (eds.) Network Processor Design: Issues and Practices, vol. 3, pp. 99–118. Morgan Kaufmann, San Francisco (2005)
7. de Bruijn, W., Slowinska, A., van Reeuwijk, K., Hruby, T., Xu, L., Bos, H.: SafeCard: a Gigabit IPS on the network card. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 311–330. Springer, Heidelberg (2006)
8. Dharmapurikar, S., Krishnamurthy, P., Sproull, T.S., Lockwood, J.W.: Deep packet inspection using parallel bloom filters. IEEE Micro 24(1), 52–61 (2004)
9. Erdogan, O., Cao, P.: Hash-AV: Fast virus signature scanning by cache-resident filters. International Journal of Security and Networks 2(1/2), 50–59 (2007)

10. Ho, J.T.L., Lemieux, G.G.: PERG-Rx: a hardware pattern-matching engine supporting limited regular expressions. In: FPGA 2009: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 257–260. ACM, New York (2009)
11. Huang, N.-F., Hung, H.-W., Lai, S.-H., Chu, Y.-M., Tsai, W.-Y.: A gpu-based multiple-pattern matching algorithm for network intrusion detection systems. In: 22nd International Conference on Advanced Information Networking and Applications - Workshops, AINAW 2008, pp. 62–67 (25-28, 2008)
12. Kojm, T.: Clamav, http://www.clamav.net/
13. Kulishov, F.: DFA-based and SIMD NFA-based regular expression matching on Cell BE for fast network traffic filtering. In: SIN 2009: Proceedings of the 2nd International Conference on Security of Information and Networks, pp. 123–127. ACM, New York (2009)
14. Lin, Y.-D., Lin, P.-C., Lai, Y.-C., Liu, T.-Y.: Hardware-Software Codesign for High-Speed Signature-based Virus Scanning. IEEE Micro 29(5), 56–65 (2009)
15. Lin, Y.-D., Tseng, K.-K., Lee, T.-H., Lin, Y.-N., Hung, C.-C., Lai, Y.-C.: A platform-based SoC design and implementation of scalable automaton matching for deep packet inspection. J. Syst. Archit. 53(12), 937–950 (2007)
16. Miretskiy, Y., Das, A., Wright, C.P., Zadok, E.: Avfs: An On-Access Anti-Virus File System. In: Proceedings of the 13th USENIX Security Symposium, p. 6. USENIX Association, Berkeley (2004)
17. Moscola, J., Lockwood, J., Loui, R., Pachos, M.: Implementation of a Content-Scanning Module for an Internet Firewall. In: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, pp. 31–38 (April 2003)
18. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 3.0, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf
19. Scarpazza, D.P., Villa, O., Petrini, F.: Exact multi-pattern string matching on the cell/b.e. processor. In: CF 2008: Proceedings of the 2008 Conference on Computing Frontiers, pp. 33–42. ACM, New York (2008)
20. Sidhu, R., Prasanna, V.: Fast regular expression matching using FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2001 (2001)
21. Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., Estan, C.: Evaluating GPUs for Network Packet Signature Matching. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software (2009)
22. Song, T., Zhang, W., Wang, D., Xue, Y.: A Memory Efficient Multiple Pattern Matching Architecture for Network Security. In: INFOCOM 2008. The 27th Conference on Computer Communications, pp. 166–170. IEEE, Los Alamitos (13-18, 2008)
23. Sourdis, I., Pnevmatikatos, D.: Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In: FCCM 2004: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Washington, DC, USA, pp. 258–267. IEEE Computer Society, Los Alamitos (2004)
24. Sourdis, I., Pnevmatikatos, D.N., Vassiliadis, S.: Scalable multigigabit pattern matching for packet inspection. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16(2), 156–166 (2008)

25. Tumeo, A., Villa, O., Sciuto, D.: Efficient pattern matching on GPUs for intrusion detection systems. In: CF 2010: Proceedings of the 7th ACM International Conference on Computing Frontiers, pp. 87–88. ACM, New York (2010)
26. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 116–134. Springer, Heidelberg (2008)
27. Vasiliadis, G., Polychronakis, M., Antonatos, S., Markatos, E.P., Ioannidis, S.: Regular Expression Matching on Graphics Hardware for Intrusion Detection. In: Proceedings of 12th International Symposium on Recent Advances in Intrusion Detection (RAID) (2009)
28. Wu, C., Yin, J., Cai, Z., Zhu, E., Chen, J.: A Hybrid Parallel Signature Matching Model for Network Security Applications Using SIMD GPU. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 191–204. Springer, Heidelberg (2009)
29. Yu, F., Katz, R.H., Lakshman, T.V.: Gigabit Rate Packet Pattern-Matching Using TCAM. In: Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP 2004), Washington, DC, USA, pp. 174–183. IEEE Computer Society, Los Alamitos (October 2004)

Automatic Discovery of Parasitic Malware

Abhinav Srivastava and Jonathon Giffin

School of Computer Science, Georgia Institute of Technology, USA
{abhinav,giffin}@cc.gatech.edu

Abstract. Malicious software includes functionality designed to block discovery or analysis by defensive utilities. To prevent correct attribution of undesirable behaviors to the malware, it often subverts the normal execution of benign processes by modifying their in-memory code images to include malicious activity. It is important to find not only maliciously-acting benign processes, but also the actual parasitic malware that may have infected those processes. In this paper, we present techniques for automatic discovery of unknown parasitic malware present on an infected system. We design and develop a hypervisor-based system, Pyrenée, that aggregates and correlates information from sensors at the network level, the network-to-host boundary, and the host level so that we correctly identify the true origin of malicious behavior. We demonstrate the effectiveness of our architecture with security and performance evaluations on a Windows system: we identified all malicious binaries in tests with real malware samples, and the tool imposed overheads of only 0%–5% on applications and performance benchmarks.

1 Introduction

Malware instances exhibit complex behaviors designed to prevent discovery or analysis by defensive utilities. In addition to file system and registry changes, malicious software often subverts the normal execution of benign processes by modifying their in-memory code image (*parasitic behavior*). For example, the Conficker worm injects undesirable dynamically linked libraries (DLLs) into legitimate software [38]. In another example, the Storm worm injects code into a user-space network process from a malicious kernel driver to initiate a DDoS attack from the infected computers [23]. Parasitic behaviors help malware execute behaviors—such as spam generation, denial-of-service attacks, and propagation—without themselves raising suspicion. When analyzing a misbehaving system to identify and eradicate malware, it is important both to terminate maliciously-acting but benign processes and to find other software that may have induced the malicious activity.

The visible effects of current attacks against software regularly manifest first as suspicious network traffic. This is due to the monetary gains involved in controlling large networks for botnet, spam, and denial of service attacks [36]. After detecting malicious traffic, network intrusion detection systems (NIDSs) can pinpoint a host within a network or enterprise responsible for that traffic [11, 25]. These network sensors can identify an infected system's IP address,

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 97–117, 2010.

network ports, and traffic behaviors. This coarse-grained information permits only coarse-grained responses: an administrator could excise an infected system from the network, possibly for reimaging. Unfortunately, in many common use scenarios, complete disk sanitization results in intolerable losses of critical data not stored elsewhere, even though that data may not have been affected by the infection. On-host analysis changes these brutal remediation techniques by providing a means to appropriately attribute malicious behavior to malicious software. Realizing that a process sending or receiving attack traffic is a hijacked benign program requires cooperation between network sensors and on-host execution monitors. By gaining a better understanding of a malware infection on a system, we can offer opportunities for surgical response.

This paper presents techniques and a prototype system, Pyrenée, for automatic discovery of unknown parasitic malware present on an infected system. Pyrenée correlates network-level events with host-level activities, so it applies exclusively to attacks that send or receive detectably-suspicious traffic. We design Pyrenée to effectively detect parasitic behavior occurring both at the user and kernel level. To remain tamper-resistant from kernel-level malware, we make use of hypervisors or virtual machine monitors (VMMs). Pyrenée's architecture is comprised of sensors at the network-to-host boundary (*network attribution sensor*), the host level (*host attribution sensor*) and the network level (*network sensor*), as well as a correlation engine that uses information provided by the sensors to identify likely malware on an infected computer.

The sensors cooperate so that Pyrenée can correctly attribute undesirable behaviors to a malicious software infection. Pyrenée uses off-the-shelf network sensors, such as BotHunter [11] or Snort [31], to detect suspicious network traffic and identify hosts with possible malware infections. When a network sensor detects malicious packets, it informs the network-to-host boundary (network attribution) sensor, deployed at the host in a trusted virtual machine (VM). On receiving the information, the network attribution sensor performs secure virtual machine introspection (VMI) to find the process bound to the malicious connection inside the guest VM. Though knowing the end-point of a malicious connection on an infected system significantly reduces the cleaning effort of an administrator, this information is still not complete. The network attribution sensor does not know if the identified process is malicious, or if it is a hijacked benign program victim of the parasitic behavior.

To find the true origin of malicious parasitic behaviors, Pyrenée uses a host-attribution sensor implanted inside the hypervisor. A process can suffer from parasitic behaviors either from another process or an untrusted kernel driver. To counter that, the host-attribution sensor monitors the execution of both user-level processes and untrusted kernel drivers. The host-attribution sensor monitors system calls and their parameters invoked by processes to detect the process-to-process parasitic behavior. To detect untrusted drivers' parasitic DLL and thread injection behaviors, we contain untrusted drivers in an isolated address space from the kernel. This design provides the host-attribution sensor an

ability to monitor kernel APIs invoked by untrusted drivers and enables it to detect parasitic behaviors originating from the untrusted drivers.

The correlation engine gathers information from all the sensors to identify the true origin of parasitic behaviors. Correlating network information with host information is a key design feature of our system. Taken alone, either approach will have diminished utility in the presence of typical attacks or normal workloads. Network-based detection can identify an infected system but cannot provide finer-grained process-specific information. Host-based detection can identify occurrences of parasitism, but it cannot differentiate malicious parasites from benign symbiotes. For example, debugging software and other benign software, such as the Google toolbar, use DLL injection for legitimate purposes. These observations are critical: A process sending or receiving malicious network traffic may not itself be malware, and a process injecting code into another process may not be malicious. Only by linking injection with subsequent malicious activity observed at the network (or other) layer can we correctly judge the activity at the host.

This paper makes the following contributions:

- We develop a well-reasoned malware detection architecture that finds unknown malware based on its undesirable network use. Our design correlates activity on the network with behaviors at the infected host.
- We correctly attribute observed behaviors to the actual malware responsible for creating those behaviors, even in the presence of parasitic malware that injects code into benign applications. Proper attribution creates the foundation for subsequent surgical remediation of the malware infection.
- Our system works for both the user- and kernel-level malware. To monitor parasitic behaviors at the user-level, we monitor system calls. For kernel-level parasitism, we securely monitor kernel APIs invoked by untrusted drivers.
- Our design satisfies protection and performance goals. We leverage virtualization to isolate security software from the infected Windows system. Our security evaluation shows that our system is able to detect the true origin of parasitic behavior occurring at user or kernel level. The performance evaluation demonstrates that even with runtime on-host monitoring, our performance impact remains only 5% or better.

2 Related Work

Pyrenée discovers unknown parasitic malware by identifying the true origin of malicious activities. To achieve its goals, it combines information gathered at both the host and the network level. Previous research in both individual areas has developed a collection of solutions to aspects of this problem.

Host-based security software generally either scans unknown programs for patterns that match signatures of known malware [17, 2] or continually monitors behaviors of software searching for unusual or suspicious runtime activity [10, 12, 32]. Pyrenée's host attribution sensor is closest in spirit to the latter

systems. It monitors the execution behavior of processes and untrusted drivers to identify instances of DLL injection or remote thread injection. Unlike traditional host-based utilities, it does not rely on injection alone as evidence of malware, as benign software sometimes uses injection for benign purposes. A heuristic-based malware detection system that monitors system calls or kernel APIs and detects code injection attacks may produce false positives. For example, DLL injection is used by the Microsoft Visual Studio debugger to monitor processes under development. Likewise, the Google toolbar injects code into `explorer.exe` (the Windows graphical file browser) to provide Internet search from the desktop. Pyrenée uses system-call information only when a network-sensor provides corroborating evidence of an attack.

Pyrenée uses virtualization to isolate its on-host software from an infected system. Virtualization has been used previously in the development of security software, including intrusion detection systems [8, 14, 20, 15, 28], firewalls [35], protection [26, 30, 40], and other areas [6]. Pyrenée's network attribution sensor is an evolution of the VMwall virtualization-based firewall design [35]. VMwall required packet queuing that introduced delay into network communication; our sensor has no such need and allows network communication to operate at full speed. The sensor makes use of virtual machine introspection (VMI), proposed by Garfinkel and Rosenblum [8], to attribute network communication to processes. Nooks [37] and SIM [33] proposed address space partitioning to isolate drivers and security applications, respectively. Pyrenée also uses address space partitioning to isolate only untrusted drivers from the core kernel and trusted drivers in a different address space.

Backtracker [19] reconstructs the sequence of steps that occurred in an intrusion by using intrusion alerts to initiate construction of event dependency graphs. In a similar way, Pyrenée uses NIDS alerts to initiate discovery of malicious software even in the presence of parasitic behaviors. Technical aspects of Backtracker and Pyrenée differ significantly. Backtracker identifies an attack's point of entry into a system by building dependencies among host-level events. It assumes that operating system kernels are trusted and hence monitors system calls; it stores each individual system call in its log for later dependency construction. Pyrenée identifies software components responsible for a post-infection attack behavior visible on the network by correlating behaviors at both the network level and host level. On the host, it monitors and stores only high-level parasitic behaviors. It does not trust the OS kernel and assumes that kernel-level malware may be present, and it monitors both system calls and kernel APIs to detect both user- and kernel-level parasitism. Both Backtracker and Pyrenée are useful to remediation in different ways: Pyrenée's information guides direct removal of malicious processes, while Backtracker's information helps develop patches or filters that may prevent future reinfection at the identified entry point.

Malware analysis tools [41] have also built upon virtualization. Dinaburg et al. [5] developed an analysis system that, among other functionality, traced the execution of system calls in a manner similar to our host attribution sensor. Martignoni et al. [21] proposed a system that builds a model of high-level malware

behavior based upon observations of low-level system calls. Like that system, Pyrenée uses a high-level characterization of DLL and thread injection identified via low-level system-call monitoring; however, our system does not employ the performance-costly taint analysis used by Martignoni. In contrast to analysis systems, our goal is to provide malware detection via correct attribution of malicious behavior to parasitic malware. We expect that it could act as a front-end automatically supplying new malware samples to deep analyzers.

3 Parasitic Malware

Pyrenée discovers parasitic malware. In this section, we present the threat model under which Pyrenée operates and describe common parasitic behaviors exhibited by malware.

3.1 Threat Model

We developed Pyrenée to operate within a realistic threat model. We assume that attackers are able to install malicious software on a victim computer system at both the user and kernel levels. Installed malware may modify the system to remain stealthy. These facts are demonstrated by recent attacks happening at the user and the kernel level. A preventive approach that does not allow users to load untrusted drivers may not be effective because users sometimes unknowingly install untrusted drivers for various reasons, such as gaming or adding new devices. Due to these reasons, we distinguish between trusted and untrusted drivers and isolate untrusted drivers in a separate address space. We assume that the malware will at some point send or receive network traffic that network-level intrusion detection systems (network sensors) are able to classify as malicious or suspicious: this may include traffic related to spam, denial-of-service attacks, propagation, data exfiltration, or botnet command-and-control.

Pyrenée makes use of virtual machine introspection (VMI) in its network attribution sensor. We perform VMI from a high-privilege virtual machine different than the infected system and assume that the high-privilege machine and the underlying hypervisor are within the trusted computing base. VMI requires kernel data structure invariants to hold. Pyrenée does not protect these data structures, but rather assumes that either existing invariant testing solutions protect the structures [27,1,34] or introspection is performed in a secure way [3]. We do not attempt to detect illicit hooking, control data attacks, evasion from hypervisor-based monitors, or modification of binaries on disk, as previous research has already studied those threats [28, 40, 18, 5].

3.2 Malware Behaviors

Parasitic malware alters the execution behavior of existing benign processes as a way to evade detection. These malware often abuse both Windows user and

Table 1. Different parasitic behavior occurring from user- or kernel-level

Number	*Source*	*Target*	*Description*
Case 1A	Process	Process	DLL and thread injection
Case 1B	Process	Process	Raw code and thread injection
Case 2A	Kernel driver	Process	DLL and thread alteration
Case 2B	Kernel driver	Process	Raw code and thread alteration
Case 2C	Kernel driver	Process	Kernel thread injection

kernel API functions to induce parasitic behaviors. We consider a malware parasitic if it injects either executable code or threads into other running processes. The parasitic behaviors can originate either from a malicious user-level process or a malicious kernel driver. Table 1 lists the different cases in which malware can induce parasitic behavior, and the following section explains each of those cases in detail.

Case 1A: *Dynamically-linked library (DLL) injection* allows one process to inject entire DLLs into the address space of a second process [29]. An attacker can author malicious functionality as a DLL and produce malware that injects the DLL into a victim process opened via the Win32 API call `OpenProcess` or created via `CreateProcess`. These functions return a process handle that allows for subsequent manipulation of the process. The malware next allocates memory inside the victim using the `VirtualAllocEx` API function and writes the name of the malicious DLL into the allocated region using `WriteProcessMemory`. Malware cannot modify an existing thread of execution in the victim process, but it can create a new thread using `CreateRemoteThread`. The malware passes to that function the address of the `LoadLibrary` API function along with the previously written-out name of the malicious DLL.

Case 1B: A *raw code injection* attack is similar to a DLL injection in that user-space malware creates a remote thread of execution, but it does not require a malicious DLL to be stored on the victim's computer system. The malware allocates memory space as before within the virtual memory region of the victim process and writes binary code to that space. It then calls `CreateRemoteThread`, passing the starting address of the injected code as an argument.

Case 2A: A kernel-level malicious driver also shows parasitic behavior by injecting malicious DLLs inside the user-space process. A malicious driver can perform this task in a variety of ways, such as by calling system call functions directly from the driver. A stealthy technique involves Asynchronous Procedure Calls (APCs): a method of executing code asynchronously in the context of a particular thread and, therefore, within the address space of a particular process [22]. Malicious drivers identify a process, allocate memory inside it, copy the malicious DLL to that memory, create and initialize a new APC, alter an existing thread of the target process to execute the inserted code, and queue the APC to later run the thread asynchronously. This method is stealthy as APCs

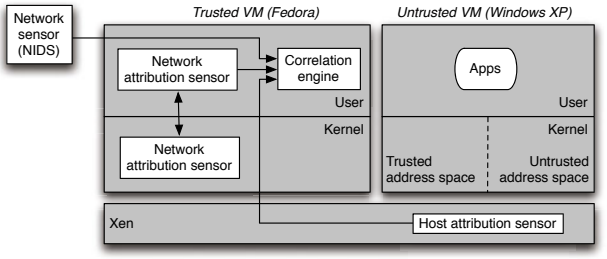


Fig. 1. Architecture of Pyrenée

are very common inside the Windows kernel, and it is very hard to distinguish between benign and malicious APCs.

Case 2B: This method is similar to the one explained in Case 2A. The difference lies in the form of malicious code that injected into a benign process. Here, malicious kernel drivers inject raw code into a benign process and execute it using the APC.

Case 2C: Finally, a *kernel thread injection* is the method by which malicious drivers execute malicious functionality entirely inside the kernel. A kernel thread executing malicious functionality is owned by a user-level process, though the user-level process had not requested its creation. By default, these threads are owned by the `System` process, however a driver may also choose to execute its kernel thread on behalf of any running process.

Our system adapts well as new attack information becomes available. Though the described methods are prevalent in current attacks, other means of injecting malicious code into benign software also exist. For example, `SetWindowsHookEx`, `AppInit_DLL`, and `SetThreadContext` APIs can be used for malice. Our general technique can easily encompass these additional attack vectors by monitoring their use in the system.

4 Architecture

Pyrenée automatically identifies at runtime the malicious code running on an infected system. That objective leads to the following design goals:

- **Accurate Attribution:** Pyrenée combines data from network-based and host-based sensors to avoid false positives, provide process or driver level granularity in malware identification, and to account for evasive behaviors of parasitic malware.
- **Automatic, Runtime Detection:** We design lightweight sensors that incur low overhead, allowing Pyrenée to operate at runtime. We identify malicious code without any human intervention.
- **Resist Direct and Indirect Attacks:** Pyrenée's tamper-resistant design prevents direct attack by a motivated attacker. We deploy all components of our system outside of an infected operating system.

Pyrenée has a modular design (Figure 1). Its architecture uses the hypervisor to provide isolation between our software and the infected system. To perform accurate detection and identification of malicious code, Pyrenée aggregates information collected from three different sensors. A network sensor identifies inbound or outbound network traffic of suspicion; we use off-the-shelf network intrusion detection systems (NIDS) like BotHunter, Snort, or Bro and will not further discuss this component. The network attribution and host attribution sensors are software programs executing in the isolated high-privilege virtual machine and hypervisor, respectively. A correlation engine, also running in the trusted VM, takes data from all three types of sensors and determines the malicious software present in the victim. Our sensors are lightweight and suitable for online detection. The following sections describe the network attribution and host attribution sensors as well as the correlation engine.

4.1 Network Attribution Sensor

The network attribution sensor maps network-level packet information to host-level process identities. Given a network sensor (NIDS) alert for some suspicious traffic flow, the network attribution sensor is responsible for determining which process is the local endpoint of that flow in the untrusted VM. This process may be malicious, or it may be a benign process altered by a parasitic malware infection. We deployed the network attribution sensor in a trusted virtual machine. It has two subcomponents: one in the VM's kernel space and one in userspace. The kernel component provides high-performance packet filtering services by intercepting both inbound and outbound network packets for an untrusted VM. The userspace component performs virtual machine introspection (VMI) whenever requested by the kernel component.

The kernel component identifies separate TCP traffic flows. Whenever it receives a SYN packet, it extracts both the source and destination IP addresses and ports, which it then passes to the userspace component for further use. The kernel component is a passive network tap and allows all packets flows to continue unimpeded. Though in the current prototype of Pyrenée we only work with TCP flows, our system is able to intercept packets of any protocol.

The userspace component determines which process in the victim VM is the local endpoint of the network flow. When invoked by the kernel component, it performs memory introspection of the untrusted VM to identify the process bound to the source (or destination) port as specified in the data received by the kernel component. To find a process name, it must locate the guest kernel's data structures that store network and process information. We have reverse engineered part of the Windows kernel to identify these structures, discussed in-depth in Section 5. The userspace component stores the extracted process and network connection information in a database to be used later by the correlation engine. The stored information helps even in the case when malware exits after sending malicious packets.

The network attribution sensor's task is to determine the end-point of a network connection originated from the guest VM. Recent kernel-level attacks

complicate this task. For example, srizbi [16] is a kernel-level bot that executes entirely in the kernel. When untrusted drivers send/receive packets from the kernel, there is no user-space process that can be considered as the end-point of the connection. Pyrenée solves this problem by monitoring the execution of untrusted drivers. Since all kernel threads created by drivers are always assigned to a user-level process, that process becomes the end-point of the in-driver connection. To determine the actual driver, we enumerate all threads of the end-point process and match against threads of untrusted drivers.

4.2 Host Attribution Sensor

The local endpoint of a malicious network flow may itself be a benign program: it may have been altered at runtime by a DLL or thread injection attack originating at a different parasitic malware process or driver. The host attribution sensor, deployed within the hypervisor, identifies the presence of possible parasitic malicious code. We describe the monitoring of both user and kernel level parasitic behaviors in the following sections.

User-level Parasitism. User-level parasitism occurs when a malicious user process injects a DLL or raw code along with a thread into a running benign process as explained in Cases 1A and 1B. To detect a process-to-process parasitic behavior, the host attribution sensor continuously monitors the runtime behavior of all processes executing within the victim VM by intercepting their system calls [9, 7]. Note that we monitor the *native API*, or the transfers from userspace to kernel space, rather than the Win32 API calls described in Section 3. High-level API monitors are insecure and may be bypassed by knowledgeable attackers, but native API monitoring offers complete mediation for user-level processes.

The host attribution sensor intercepts all system calls, but it processes only those that may be used by a DLL or thread injection attack. This list includes `NtOpenProcess`, `NtCreateProcess`, `NtAllocateVirtualMemory`, `NtWriteVirtualMemory`, `NtCreateThread`, and `NtClose`, which are the native API forms of the higher-level Win32 API functions described previously. The sensor records the system calls' parameter values: *IN* parameters at the entry of the call and *OUT* parameters when they return to userspace. Recovering parameters requires a complex implementation that we describe in detail in Section 5.

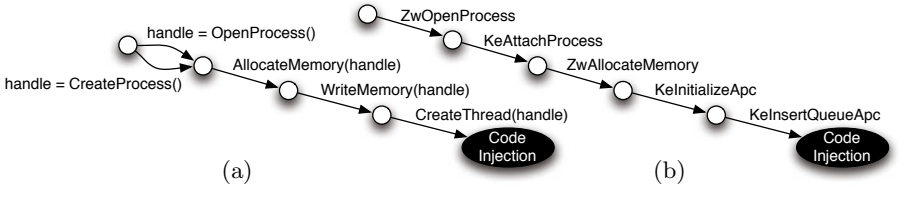


Fig. 2. Runtime parasitic behavioral models. (a) Process-to-process injection. (b) Driver-to-process injection.

The sensor uses an automaton description of malware parasitism to determine when DLL or thread injection occurs. The automaton (Figure 2a) characterizes the series of system calls that occur during an injection. As the sensor intercepts system calls, it verifies them against an instance of the automaton specific to each possible victim process. We determine when the calls apply to the same victim by performing data-flow analysis on the process handle returned by `NtOpenProcess` and `NtCreateProcess`. Should the handle be duplicated (using `NtDuplicateObject`), we include the new handle in further analysis. The sensor communicates information about detected injections to the correlation engine for further use.

Kernel-level Parasitism. Kernel-level parasitism occurs when a malicious kernel driver injects either a DLL or raw code followed by the alteration of an existing targeted process' thread (Case 2A and 2B). A kernel-level malicious driver can also create a new thread owned by any process as explained in Case 2C. To detect kernel-to-process parasitic behavior, the host attribution sensor monitors all kernel APIs invoked by untrusted drivers. However, there is no monitoring interface inside the kernel for drivers similar to the system-call interface provided to user applications. To solve this problem, Pyrenée creates a monitoring interface inside the kernel for untrusted drivers by isolating them in a separate address space and monitoring kernel APIs invoked by untrusted drivers through this new interface.

Pyrenée creates a new address space inside the hypervisor transparent to the guest OS and loads all untrusted drivers in this address space. This new address space is analogous to the existing kernel address space, however permissions are set differently. The existing kernel space, called the *trusted page table* (`TPT`), contains all the core kernel and trusted driver code with read and execute permissions, and untrusted driver code with read-only permissions. The untrusted driver address space, called the *untrusted page table* (`UPT`), contains untrusted code with read and execute permissions, and trusted code as non-readable, non-writable, and non-executable. Pyrenée also makes sure that the data pages mapped in both the address spaces are non-executable. Table 2 shows the permissions set on UPT and TPT memory pages. With these permission bits, any control flow transfers from untrusted to trusted address space induce page faults thereby enabling the host-attribution sensor to monitor kernel APIs invoked by untrusted drivers.

Pyrenée differentiates between trusted and untrusted drivers at the time of loading to decide in which address space they must be mapped. This differentiation can be made using certificates. For example, a driver signed by Microsoft

Table 2. Permission bits on trusted and untrusted address spaces

Address Space	*Trusted Code*	*Trusted Data*	*Untrusted Code*	*Untrusted Data*
Trusted	rx	rw	r	rw
Untrusted	—	rw	rx	rw

can be loaded in the trusted address space. However, Microsoft might not rely on drivers signed by other parties whose authenticity is not verified. With this design, all Microsoft signed drivers are loaded into the trusted address space, and other drivers signed by third party vendors or unsigned drivers, including kernel malware, are loaded into the untrusted address space.

Due to the isolated address space, the host-attribution sensor intercepts all kernel APIs invoked by untrusted drivers and inspects their parameters. The sensor uses an automaton to characterize the parasitic behavior originating from malicious drivers. When the sensor intercepts kernel APIs, it verifies against the automaton to recognize the parasitic behavior. In our current prototype, we create an automaton based on the kernel APC-based code injection (Figure 2b). The host-attribution sensor records the gathered information for future use by the correlation engine.

4.3 Correlation Engine

The correlation engine identifies which code on an infected system is malicious based on information from our collection of sensors [39]. The engine has three interfaces that communicate with a NIDS, the host attribution sensor, and the network attribution sensor. Architecturally, it resides in the isolated, high-privilege VM.

The NIDS provides network alert information to the correlation engine's first interface. This information includes the infected machine's IP address, port used in the suspicious flow, and other details. The alert acts as a trigger that activates searches across information from the software sensors. The second interface gathers information from the network attribution sensor, which provides information that maps the malicious network connection identified by the NIDS to a host-level process.

The third interface collects information from the host attribution sensor. In its process-to-process injection report, the host attribution sensor passes identifiers of injecting and victim processes, a handle for the victim of the injection, and other data. When receiving this information, the correlation engine uses VMI to retrieve detailed data about the victim and injecting processes, including their name, their component DLLs, and their open files. Should the victim process not have an identifier, as is the case for victims created via `NtCreateProcess`, the engine uses the victim's process handle to recover information about the victim. Section 5 provides low-level details of this data extraction. In the kernel-to-process injection report, it passes details of the victim process, such as the process identifier, name, and the name of the malicious kernel driver.

Based on the information provided by sensors, the correlation engine constructs a list of malicious processes and drivers. It matches attack information provided by a NIDS with network flow endpoint records generated by the network attribution sensor. When it finds a match, it extracts the name of the process bound to the malicious connection. Using information from the host attribution sensor, it determines whether or not the process has suffered from a parasitic attack. When it finds an injection, it extracts the name of the injecting

process or driver and adds it to the list of malicious code. Finally, it identifies other benign processes infected by the malware by searching again within the host attribution sensor's records. The correlation engine periodically purges records generated by the network and host attribution sensors.

5 Low-Level Implementation Details

Pyrenée is an operating prototype implemented for Windows XP SP2 victim systems hosted in virtual machines by the Xen hypervisor version 3.2. The high-privilege VM executing our software runs Fedora Core 9. Implementing Pyrenée for Windows victim systems required technical solutions to challenging, low-level problems.

5.1 Fast Network Flow Discovery

The network attribution sensor intercepts all inbound and outbound network flows of untrusted virtual machines. To intercept packets at a fast rate, we deployed the sensor's packet filter inside the trusted VM's kernel-space; we developed the packet filter as a Linux kernel module. To capture packets before they exit the network, we set up our untrusted VMs to use a virtual network interface bridged to the trusted VM. We inserted a hook into a bridge-based packet filtering framework called *ebtables* [4] to view packets crossing the bridge. Whenever the sensor's kernel component receives a TCP SYN packet from the hook, it notifies the userspace component to perform introspection.

5.2 Introspection

The network attribution sensor identifies local processes that are the endpoints of network flows via virtual machine introspection. This requires the sensor's userspace component to inspect the runtime state of the victim system's kernel

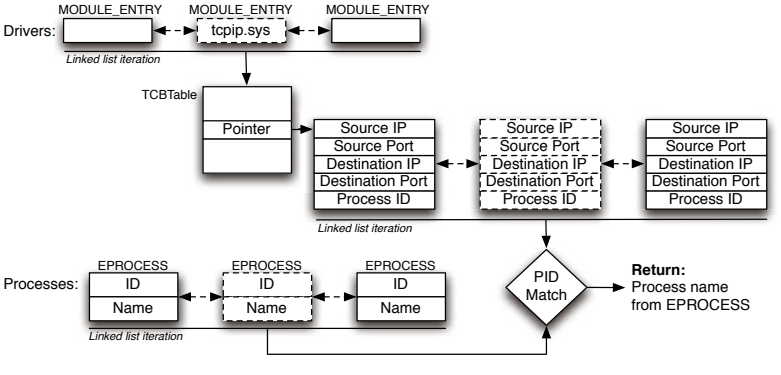


Fig. 3. Network connection to host-level process correlation in Windows

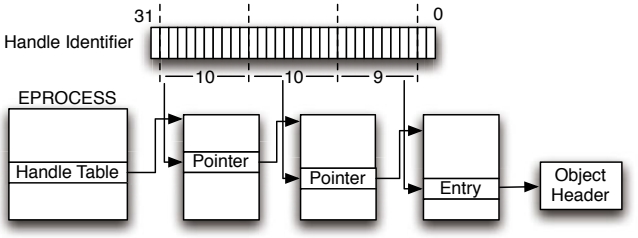


Fig. 4. Handle resolution in Windows converts a 32-bit handle identifier into a structure for the object referenced by the handle. Resolution operates in a manner similar to physical address resolution via page tables.

state. Unfortunately, Windows does not store network port and process name information in a single structure. A network driver (`tcpip.sys`) manages network connection related information. To locate the data structure corresponding to `tcpip.sys`, the sensor's userspace component iterates across the kernel's list of loaded drivers to find the structure's memory address. The driver maintains a pointer to a structure called `TCBTable`, which in turn points to a linked list of objects containing network ports and process IDs for open connections. To convert the process ID to a process name, the component iterates across the guest kernel's linked list of running processes. Figure 3 illustrates the complete process of resolving a network connection to a host-level process name.

The correlation engine uses VMI across handle tables to identify the names of processes that receive DLL or thread injection from other, potentially malicious, software. The engine knows handle identifiers because the host attribution sensor observes *IN* parameters to the Windows system calls used as part of an injection, and these parameters include handles. All handles used by a process are maintained by the Windows kernel in handle tables, which are structured as shown in Figure 4.

To resolve a handle to a process name, the correlation engine uses the handle to find the corresponding `EPROCESS` data structure in the Windows kernel memory. Since it knows the process ID of an injecting process, the engine can find that process' handle table. It searches the table for the specific object identifier recorded by the host attribution sensor. As a pleasant side-effect, this inspection of the handle table will additionally reveal the collection of files and registries currently open to the possibly malicious injecting process.

5.3 System Call Interpositioning and Parameter Extraction

Pyrenée's host attribution sensor requires information about system calls used as part of DLL or thread injection. We developed a system call interpositioning framework deployable in Xen; this framework supports inspection of both *IN* and *OUT* system call parameters. An *IN* parameter's value is passed by the caller of a system call while an *OUT* parameter's value is filled after the execution of the system call inside the kernel.

Windows XP uses the fast x86 system-call instruction `SYSENTER`. This instruction transfers control to a system-call dispatch routine at an address specified in the `IA32_SYSENTER_EIP` register. Unfortunately, the Intel VTx hardware virtualization design does not allow the execution of `SYSENTER` to cause a VM to exit out to the hypervisor. As a result, our host attribution sensor must forcibly gain execution control at the beginning of a system call. It alters the contents of `IA32_SYSENTER_EIP` to contain a memory address that is not allocated to the guest OS. When a guest application executes `SYSENTER`, execution will fault to the hypervisor, and hence to our code, due to the invalid control-flow target.

Inside the hypervisor, the sensor processes all faults due to its manipulation of the register value. It records the system call number (stored in the `eax` register), and it uses the `edx` register value to locate system-call parameters stored on the kernel stack. The sensor extracts *IN* parameters with a series of guest memory read operations. It uses the `FS` segment selector to find the *Win32 thread information block* (TIB) containing the currently-executing process' ID and thread ID. It then modifies the instruction pointer value to point at the original address of the system-call dispatch routine and re-executes the faulted instruction.

We use a two-step procedure to extract values of *OUT* parameters at system-call return. In the first step, we record the value present in an *OUT* parameter at the beginning of the system call. Since *OUT* parameters are passed by reference, the stored value is a pointer. In order to know when a system call's execution has completed inside the kernel, we modify the return address of an executing thread inside the kernel with a new address that is not assigned to the guest OS. This modification occurs when intercepting the entry of the system call. In the second step, a thread returning to usermode at the completion of a system call will fault due to our manipulation. As before, the hypervisor receives the fault. Pyrenée reads the values of *OUT* parameters, restores the original return address, and re-executes the faulting instruction. By the end of the second step, the host attribution sensor has values for both the *IN* and *OUT* system-call parameters.

5.4 Address Space Construction and Switching

We create isolated address space for untrusted drivers using the Xen hypervisor and the Windows XP 32-bit guest operating system, though our design is general and applicable to other operating systems and hypervisors. We allocate memory for UPT page tables transparent to the guest OS inside the hypervisor. We then map untrusted driver code pages into the UPT and trusted kernel and driver code into the TPT. We mark all untrusted driver code pages in TPT as non-executable and non-writable and mark all trusted code pages in UPT as non-executable, non-writable, and non-readable.

Pyrenée switches between the two address spaces depending upon the execution context. It manipulates the `CR3` register: a hardware register that points to the current page tables used by memory management hardware and inaccessible to any guest OS. When an untrusted driver invokes a kernel API, execution faults into the hypervisor due the non-executable kernel code in the UPT. Inside

the hypervisor, Pyrenée verifies the legitimacy of the control flow by checking whether the entry point into the TPT is valid. If the entry point is valid, it switches the address space by storing the value of `TPT_CR3`, the trusted page table base, into `CR3`. If the entry point is not valid, Pyrenée records this behavior as an attack and raises an alarm. Similarly, control flow transfers from TPT to UPT fault because untrusted driver code pages are marked non-executable inside the TPT. On this fault, Pyrenée switches the address space by storing the untrusted page table base, `UPT_CR3`, in the `CR3` register.

Pyrenée identifies the legitimate entry points into the TPT by finding the kernel and trusted drivers' exported functions. These exported functions' names and addresses are generated from the PDB files available from Microsoft's symbol server. Pyrenée keeps this information in the hypervisor for the host-attribution sensor.

5.5 Interception of Driver Loading

Pyrenée requires knowledge of drivers' load addresses to map their code pages into either the UPT or TPT. Since Windows dynamically allocates memory for all drivers, these addresses change. Moreover, Windows uses multiple mechanisms to load drivers. Pyrenée intercepts all driver loading mechanisms. It rewrites the kernel's binary code on driver loading paths automatically at runtime. It modifies the direct call instruction to the `ObInsertObject` kernel function by changing its target to point to a location in the guest which is not assigned to the guest VM; it stores the original target. With this design, during the driver loading process execution faults into the hypervisor. On the fault, Pyrenée extracts the driver's load address securely from the driver object and resumes the execution at the original target location. This design provides complete interpositioning of driver loading.

6 Evaluation

We tested our prototype implementation of Pyrenée to evaluate its ability to appropriately identify malicious software on infected systems, its performance, and its avoidance of false positives. To generate alerts notifying the correlation engine of suspicious network activity in our test environment, we ran a network simulator that acted as a network-based IDS.

6.1 User-Level Malware Identification

We tested Pyrenée's ability to detect process-to-process parasitic behaviors with the recent `Conficker` worm [38]. Conficker employs DLL injection to infect benign processes running on the victim system. We executed Conficker inside a test VM monitored by Pyrenée and connected to a network overseen by our NIDS simulator. When executed, the worm ran as a process called `rundll32.exe`. The

host attribution sensor recorded DLL injection behavior from `rundll32.exe` targeting specific `svchost` processes.

When our NIDS simulator sent the IP addresses and port numbers for outbound malicious traffic to Pyrenée's correlation engine, the engine then determined what malicious code on the host was responsible. It searched the network attribution sensor's data to extract the name of the process bound to the connection's source port, here `svchost.exe`. It then searched the host attribution sensor's data and found that `svchost.exe` was the victim of a parasitic DLL injection from `rundll32.exe`. The correlation engine also found the names of other executables infected by the malware, and it generated a complete listing that could be sent to a security administrator.

We repeated these tests with the `Adclicker.BA` trojan and successfully detected its parasitic behavior.

6.2 Kernel-Level Malware Identification

We evaluated Pyrenée's ability to detect kernel-level parasitism by testing it with the recent Storm worm [23]. Storm is kernel-level malware that exhibits parasitic behaviors by injecting malicious DLLs into the benign `services.exe` process, causing `services.exe` to launch DDoS attacks. We loaded Storm's malicious driver in the test VM. Since the driver is untrusted, Pyrenée loaded it into the separate isolated address space. On the execution of the driver's code, all kernel APIs invoked by the driver were verified and logged by Pyrenée's host attribution sensor. The sensor found that the driver was performing injection via APCs, and it recorded both the parasitic behavior and the victim process.

When our network simulator flagged the traffic made by `services.exe`, the correlation engine gathered the data collected by the host and network attribution sensors. The network attribution sensor determined services.exe to be the end-point of the connection, and the host attribution sensor identified the parasitism of the malicious driver.

6.3 Performance

We designed Pyrenée to operate at runtime, so its performance cost on an end user's system must remain low. We tested our prototype on an Intel Core 2 Quad 2.66 GHz system. We assigned 1 GB of memory to the untrusted Windows XP SP2 VM and 3 GB combined to the Xen hypervisor and the high-privilege Fedora Core 9 VM. We carried out CPU and memory experiments using a Windows benchmark tool called `PassMark Performance Test` [24]. We measured networking overheads using `IBM Page Detailer` [13] and `wget`. Our experiments measured Pyrenée's overhead during benign operations, during active parasitic attacks, and during the isolation of a heavily-used driver in the UPT. We executed all measurements five times and present here the median values.

First, we measured Pyrenée's overhead on CPU-bound and memory intensive operations. Tables 3 and 4 list a collection of benchmark measurements for

Table 3. Results of CPU performance tests for unmonitored execution and for Pyrenée's monitoring with and without parasitic behaviors present; higher absolute measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Parasitic Behavior*			
		Present	%	*Absent*	%
Integer Math (MOps/sec)	126.5	92.5	26.88	124.8	1.34
Floating Point Math (MOps/sec)	468.4	439.5	6.17	444.3	5.14
Compression (KB/sec)	1500.9	1494.7	0.41	1496.0	0.32
Encryption (MB/sec)	4.21	4.19	0.48	4.20	0.24
String Sorting (Thousand strings/sec)	1103.3	1072.2	2.82	1072.3	2.81

Table 4. Results of memory performance tests for unmonitored execution and for Pyrenée's monitoring with and without parasitic behaviors present; higher absolute measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Parasitic Behavior*			
		Present	%	*Absent*	%
Allocate Small Block (MB/sec)	2707.4	2322.3	14.22	2704.1	0.12
Write (MB/sec)	1967.0	1931	1.83	1942.9	1.23

execution in a VM with and without Pyrenée's monitoring. For executions including Pyrenée, we measured performance both during execution of a DLL injection attack against an unrelated process and during benign system operation. Our system's performance in the absence of parasitic behavior is excellent and largely reflects the cost of system-call tracing. Experiments including the execution of an injection attack show diminished performance that ranges from inconsequential to a more substantial performance loss of 27%. The additional overhead measured during the attack occurred when Pyrenée's host sensor identified injection behavior and harvested state information for its log. This overhead is infrequent and occurs only when parasitic behaviors actually occur.

Next, we measured Pyrenée's performance during network operations. Using the *IBM Page Detailer*, we measured the time to load a complex webpage (`http://www.cnn.com`) that consisted of many objects spread across multiple servers. The page load caused the browser to make numerous network connections—an important test because Pyrenée's network attribution sensor intercepts each packet and performs introspection on SYN packets. The result, shown in Table 5, demonstrates that the overhead of the network attribution sensor is low. We next executed a network file transfer by hosting a 174 MB file on a local networked server running `thttpd` and then downloading the file over HTTP using `wget` from the untrusted VM. Table 5 shows that Pyrenée incurred less than 3% overhead on the network transfer; we expect that this strong performance is possible because its packet interception design does not require it to queue and delay packets.

Finally, we measured the cost of our driver isolation strategy by isolating a heavily-used driver in the UPT, forcing a high volume of page faults handled

Table 5. Results of the network performance tests for unmonitored execution and for Pyrenée's monitoring without parasitic behaviors present; smaller measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Pyrenée*	%
Page Loading (sec)	3.64	3.82	4.95
Network File Copy (sec)	38.00	39.00	2.63

Table 6. Effect of isolating the tcpip.sys driver on CPU operations for unmonitored execution and for Pyrenée's monitoring without parasitic behaviors present; higher measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Pyrenée*	%
Integer Math (MOps/sec)	126.5	122.0	3.55
Floating Point Math (MOps/sec)	468.4	434.8	7.17
Compression (KB/sec)	1500.9	1467.5	2.23
Encryption (MB/sec)	4.21	4.11	2.38
String Sorting (Thousand strings/sec)	1103.3	1060.8	3.85

Table 7. Effect of isolating the tcpip.sys driver on memory performance for unmonitored execution and for Pyrenée's monitoring without parasitic behaviors present; higher measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Pyrenée*	%
Allocate Small Block (MB/sec)	2707.4	2649.8	2.12
Write (MB/sec)	1967.0	1922.0	2.29

Table 8. Effect of isolating the tcpip.sys driver on network performance for unmonitored execution and for Pyrenée's monitoring without parasitic behaviors present; smaller measurements are better. Percentages indicate performance loss.

Operations	*Unmonitored*	*Pyrenée*	%
Network File Copy (sec)	38.00	51.00	34.21

by our hypervisor-level code. We isolated the networking driver `tcpip.sys` and repeated our previous CPU, memory, and network performance measurements in the new setting without active parasitic behaviors. We anticipated that CPU and memory overheads would remain similar, but that network operations would experience decreased performance. Tables 6, 7, and 8 provide evidence that our intuition was correct. Given that the moderate performance cost of isolating a driver in the UPT is borne only by operations invoking that driver's functionality, we believe that it represents a feasible deployment strategy for unknown and untrusted drivers. The clear performance gain to be had by relocating known-benign drivers in the TPT provides an incentive for driver authors to produce verifiably-safe drivers acceptable to a driver-signing authority.

6.4 False Positive Analysis

Pyrenée finds malicious code present on an infected system whenever it receives an alert from a NIDS; it does not detect attacks directly on its own. Hence, false positives will be exhibited by Pyrenée only when it identifies a benign processes' binary or a driver as malicious. We see two possible reasons for such behavior.

First, a NIDS may have false positives when distinguishing between benign and malicious traffic, and it may mis-characterize benign traffic as malicious. In this case, when the NIDS sends an alert along with the network-related information, the network attribution sensor will identify the process that is bound to the connection, and the correlation engine will mark that process as malicious. Certainly, this is a false positive. Fortunately, this problem will diminish over time as NIDS' false positive rates decrease [11]. Even in the case of such false positives, Pyrenée helps an administrator meaningfully look into the actual problem by locating the endpoint of the network traffic. We feel that this design is stronger than an alternative that stores a whitelist of benign parasitic applications and considers malicious parasitic behaviors to be those initiated by non-whitelisted applications. The alternative design requires a whitelist that may not be feasible to generate.

Second, Pyrenée could identify a benign process as malicious when a NIDS correctly generates an alert. Absent implementation bugs, this could only be possible if the network attribution sensor or the host attribution sensor collect incorrect information. Benign parasitic behaviors, such as injections caused by debugging, will not appear to be malicious unless the debugged process is using the network in a way that appears to the NIDS as an attack.

7 Conclusions

We demonstrated the usefulness of identifying malicious code present on an infected system during attacks. We presented techniques and a prototype system, Pyrenée, for the automatic discovery of unknown malicious code. Pyrenée correlates network-level events to host-level activities with the help of multiple sensors and the correlation engine. When alerted by a NIDS, our system discovered malicious code, even in the presence of parasitic malware, by correlating information gathered from the host and network attribution sensors. Real malware samples showed that Pyrenée correctly identified malicious code. Our performance analysis demonstrated that our solution was suitable for real world deployment.

Acknowledgment of Support and Disclaimer. We thank our shepherd, Davide Balzarotti, and our anonymous reviewers for their extremely helpful comments. This material is based upon work supported by National Science Foundation contract number CNS-0845309. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the NSF or the U.S. Government.

References

1. Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structures invariants. In: ACSAC, Anaheim, CA (December 2008)
2. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA (May 2005)
3. Christodorescu, M., Sailer, R., Schales, D., Sgandurra, D., Zamboni, D.: Cloud security is not (just) virtualization security. In: Cloud Computing Security Workshop, Chicago, IL (November 2009)
4. Community Developers. Ebtables, `http://ebtables.sourceforge.net/` (last accessed April 15, 2010)
5. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: ACM CCS, Alexandria, VA (October 2008)
6. Dunlap, G., King, S., Cinar, S., Basrai, M., Chen, P.: Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In: OSDI, Boston, MA (December 2002)
7. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for UNIX processes. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 1996)
8. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS, San Diego, CA (February 2003)
9. Giffin, J., Jha, S., Miller, B.: Detecting manipulated remote call streams. In: 11th USENIX Security Symposium, San Francisco, CA (August 2002)
10. Giffin, J.T., Jha, S., Miller, B.P.: Efficient context-sensitive intrusion detection. In: NDSS, San Diego, CA (February 2004)
11. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting malware infection through IDS-driven dialog correlation. In: USENIX Security Symposium, Boston, MA (August 2007)
12. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security 6(3), 151–180 (1998)
13. IBM. Ibm page detailer, `http://www.alphaworks.ibm.com/tech/pagedetailer/download` (last accessed April 15, 2010)
14. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through VMM-based 'out-of-the-box' semantic view. In: ACM CCS, Alexandria, VA (November 2007)
15. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid. In: ACM VEE, Seattle, WA (March 2008)
16. Kasslin, K.: Evolution of kernel-mode malware, `http://igloo.engineeringforfun.com/malwares/Kimmo_Kasslin_Evolution_of_kernel_mode_malware_v2.pdf` (last accessed April 15, 2010)
17. Kephart, J., Arnold, W.: Automatic extraction of computer virus signatures. In: Virus Bulletin, Jersey, Channel Islands, UK (1994)
18. Kim, G.H., Spafford, E.H.: The design and implementation of tripwire: a file system integrity checker. In: ACM CCS, Fairfax, VA (November 1994)
19. King, S.T., Chen, P.M.: Backtracking intrusions. In: ACM SOSP, Bolton Landing, NY (October 2003)
20. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: USENIX Security Symposium, San Jose, CA (August 2008)

21. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A layered architecture for detecting malicious behaviors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 78–97. Springer, Heidelberg (2008)
22. MSDN. Asynchronous procedure calls, `http://msdn.microsoft.com/en-us/library/ms681951VS.85.aspx` (last accessed April 15, 2010)
23. OffensiveComputing. Storm Worm Process Injection from the Windows Kernel, `http://www.offensivecomputing.net/?q=node/661` (last accessed April 15, 2010)
24. Passmark Software. PassMark Performance Test, `http://www.passmark.com/products/pt.htm` (last accessed April 15, 2010)
25. Paxson, V.: Bro: A system for detecting network intruders in real-time. In: Usenix Security, San Antonio, TA (January 1998)
26. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An architecture for secure active monitoring using virtualization. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2008)
27. Petroni Jr., N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In: USENIX Security Symposium, Vancouver, BC, Canada (August 2006)
28. Petroni Jr., N.L., Hicks, M.: Automated detection of persistent kernel control-flow attacks. In: ACM CCS, Alexandria, VA (November 2007)
29. Richter, J.: Load your 32-bit DLL into another process's address space using injlib. Microsoft Systems Journal 9(5) (May 1994)
30. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
31. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of USENIX LISA, Seattle, WA (November 1999)
32. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: IEEE Symposium on Security and Privacy, Oakland, CA (May 2001)
33. Sharif, M., Lee, W., Cui, W., Lanzi, A.: Secure in-vm monitoring using hardware virtualization. In: ACM CCS, Chicago, IL (November 2009)
34. Srivastava, A., Erete, I., Giffin, J.: Kernel data integrity protection via memory access control. Technical Report GT-CS-09-05, Georgia Institute of Technology, Atlanta, GA (2009)
35. Srivastava, A., Giffin, J.: Tamper-resistant, application-aware blocking of malicious network connections. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 39–58. Springer, Heidelberg (2008)
36. Staniford, S., Paxson, V., Weaver, N.: How to 0wn the internet in your spare time. In: USENIX Security Symposium, San Francisco, CA (August 2002)
37. Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the reliability of commodity operating systems. In: ACM SOSP, Bolton Landing, NY (October 2003)
38. ThreatExpert. Conficker/downadup: Memory injection model. `http://blog.threatexpert.com/2009/01/confickerdownadup-memory-injection.html` (last accessed April 15, 2010)
39. Valdes, A., Skinner, K.: Probabilistic alert correlation. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, p. 54. Springer, Heidelberg (2001)
40. Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: ACM CCS, Chicago, IL (November 2009)
41. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Security & Privacy 5(2) (March 2007)

BotSwindler: Tamper Resistant Injection of Believable Decoys in VM-Based Hosts for Crimeware Detection*

Brian M. Bowen[1], Pratap Prabhu[1], Vasileios P. Kemerlis[1], Stelios Sidiroglou[2], Angelos D. Keromytis[1], and Salvatore J. Stolfo[1]

[1] Department of Computer Science, Columbia University
{bb2281,pvp2105,vk2209,ak2052,sjs11}@columbia.edu
[2] Computer Science and Artificial Intelligence Laboratory, MIT
stelios@csail.mit.edu

Abstract. We introduce BotSwindler, a bait injection system designed to delude and detect crimeware by forcing it to reveal during the exploitation of monitored information. The implementation of BotSwindler relies upon an out-of-host software agent that drives user-like interactions in a virtual machine, seeking to convince malware residing within the guest OS that it has captured legitimate credentials. To aid in the accuracy and realism of the simulations, we propose a low overhead approach, called virtual machine verification, for verifying whether the guest OS is in one of a predefined set of states. We present results from experiments with real credential-collecting malware that demonstrate the injection of monitored financial bait for detecting compromises. Additionally, using a computational analysis and a user study, we illustrate the believability of the simulations and we demonstrate that they are sufficiently human-like. Finally, we provide results from performance measurements to show our approach does not impose a performance burden.

1 Introduction

The creation and rapid growth of an underground economy that trades in stolen digital credentials has spurred the growth of crime-driven bots that harvest sensitive data from unsuspecting users. This form of malevolent software employs a variety of techniques ranging from web-based form grabbing and key stroke logging, to screenshots and video capture for the purposes of pilfering data on remote hosts to automate financial crime [1,2]. The targets of such malware range from individual users and small companies to the most wealthiest organizations [3]—recent studies indicate that bot infections are on the rise and up to 9% of the machines in an enterprise are now bot-infected [4].

Traditional crimeware detection techniques rely on comparing signatures of known malicious instances to identify unknown samples, or on anomaly-based detection techniques in which host behaviors are monitored for large deviations from a baseline.

* This work was partly supported by the National Science Foundation through grants CNS-07-14647 and CNS-09-14312. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 118–137, 2010.

Unfortunately, these approaches suffer a large number of known weaknesses. Signature-based methods can be useful when a signature is known, but due to the large number of possible variants, learning and searching all possible signatures to identify unknown binaries is intractable [5]. On the other hand, anomaly-based methods are susceptible to false positives and negatives, limiting their potential utility. Consequently, a large amount of existing crimeware now operate undetected by antivirus software. A recent study focused of Zeus[1] (the largest botnet with over 3.6 million PC infections in the US alone [7]), revealed that the malware bypassed up-to-date antivirus software 55% of the time [8].

Another drawback of conventional host-based antivirus software is that it typically monitors from within the host it is trying to protect, making it vulnerable to evasion or subversion by malware; we see an increasing number of malware attacks that disable defenses such as antivirus software prior to undertaking some malicious activity [9].

In this work, we introduce BotSwindler, a novel system designed for the proactive detection of credential stealing malware on VM-based hosts. BotSwindler relies upon an out-of-host software agent to drive user simulations that are meant to convince malware residing within the guest OS that it has captured legitimate credentials. By the nature of its out-of-host operating position, the simulator is tamper resistant and difficult to detect by malware residing within the host environment. We posit that malware that detects BotSwindler would need to analyze the behavior of its host and decide whether it is observing a human or not. In other words, the crimeware would need to solve a Turing Test [10]. We assert that if attackers are forced to spend their time looking at the actions on each infected host one by one to determine if they are real or not in order to steal information, BotSwindler would be a success; the attackers' task does not scale. To generate simulations, BotSwindler relies on a formal language that is used to specify a simulation of human user's sequence of actions. The language provides a flexible way to generate variable simulation behaviors that appear realistic. Simulations can be tuned to mimic particular users by using various models for keystroke speed, mouse speed and the frequency of errors made during typing.

One of the challenges in designing an out-of-host simulator lies in the ability to detect the underlying state of the OS. That is, to verify the success or failure of mouse and keyboard events that are passed to the guest OS. For example, if the command is given to open a browser and navigate to a particular URL, the simulator must validate that the URL was successfully opened before proceeding with the next command. To aid in the accuracy and realism of the simulations, we developed a low overhead approach, called virtual machine verification (VMV), for verifying whether the state of the guest OS is in one of a predefined set of states.

BotSwindler aims to detect crimeware by deceptively inducing it into an observable action during the exploitation of monitored information injected into the guest OS. To entice attackers with information of value, the system supports a variety of different types of bait credentials including decoy Gmail and PayPal authentication credentials, as well as those from a large financial institution[2]. Our system automatically monitors

[1] Zeus uses key-logging techniques to steal sensitive data such as user names, passwords, account numbers. It can be purchased on the black market for $600, complete with support and maintenance [6].

[2] By agreement, the institution requested that its name be withheld.

the decoy accounts for misuse to signal exploitation and thus detect the host infection by credential stealing malware.

BotSwindler presents an instance of a system and approach that can be used to deal with information-level attacks, regardless of their origin. In our prototype, we rely on credentials for financial institutions because they are good examples that we can easily evaluate, but the approach is aimed at any kind of large-scale automated harvesting of "interesting" data — where "interesting" depends on both the environment and the malware. Although we demonstrate our system with three types of credentials, the system can be extended to support any type of credential that can be monitored for misuse. As one of the contributions of this work, we consider different applications of BotSwindler including how it could be applied practically in an enterprise environment with simulations and decoys adapted to the specific deployment setting. In part of doing so, we discuss how BotSwindler can be deployed to service hosts that include those which are not VM-based, making this approach broadly applicable.

We have implemented a prototype version of BotSwindler using a modified version of QEMU [11] running on a Linux host. User simulation is implemented using X11 libraries and interaction with the graphical frame buffer. We demonstrate our prototype through experiments with crimeware on a Windows guest, but BotSwindler can operate on any guest operating system supported by the underlying hypervisor or virtual machine monitor (VMM).

1.1 Overview of Results

To demonstrate the effectiveness of BotSwindler, we tested our prototype against real crimeware samples obtained from the wild. Our results from two separate experiments with different types of decoy credentials show that BotSwindler succeeds in detecting malware through attackers' exploitation of the monitored bait. In our first experiment with 116 Zeus samples, we received 14 distinct alerts using PayPal and Gmail decoys. In a second experiment with 59 different Zeus samples, we received 3 alerts from our banking decoys.

The long-term viability of BotSwindler defense largely depends on the believability of the bait-injecting simulations by the attackers. We performed a computational analysis to see if attackers could employ machine learning algorithms on keystrokes to distinguish simulations. We present results from experiments running Naive Bayes and Support Vector Machine (SVM) classifiers on real and generated timing data to show that they produce nearly identical classification results making this kind of analysis ineffectual for an adversary. To show that adversaries resorting to manual inspection of the user activities would be sufficiently challenged, we evaluated the believability of user simulations via a decoy Turing Test in which human judges were tasked with trying to distinguish BotSwindler's actions from those of a real human. The failure of the judges to distinguish suggests BotSwindler's simulations are convincingly human-like. In our study with 25 human judges evaluating 10 videos of BotSwindler actions and of a human, the judges' average success rate was 46%, indicating the simulations provide a good approximation of human actions.

Finally, recognizing that attackers may try to distinguish simulated behavior via performance metrics, we evaluated the overhead of our approach by measuring the cost

imposed by the virtual machine verification (VMV) technique. Our results indicate that VMV imposes no measurable overhead, making the technique difficult to detect by malware using performance analysis [12].

1.2 Summary of Contributions

This paper makes the following contributions:

- **BotSwindler architecture:** It introduces BotSwindler, a novel, accurate, efficient, and tamper-resistant zero-day crimeware detection system. BotSwindler relies on the use of decoy injection whereby bogus information is used to bait and delude crimeware, causing it to reveal itself during the exploitation of the monitored information.
- **VMSim language:** It introduces VMSim, a new language for expressing simulated user behavior. VMSim facilitates the construction and reproduction of complex user activity, including specifying aggregate statistical behavior.
- **Virtual Machine Verification (VMV):** It introduces virtual machine verification, a low overhead approach for verifying simulation state. VMV enables robust out-of-host user action simulation through graphical state verification.
- **Real malware detection results:** It presents results to show the effectiveness of BotSwindler in detecting real malware when decoy PayPal, Gmail, and banking credentials are injected, stolen, and exploited by the attackers.
- **Statistical and information theoretic analysis:** It presents the results of a computational analysis on generated keystroke timing data to show it would be difficult to detect simulations through analysis with machine learning algorithms or entropy measurements.
- **Believability user study results:** It presents user study results that show the believability of simulations created with BotSwindler's VMSim language.
- **Performance overhead results:** It shows that BotSwindler imposes no measurable overhead, hence making itself undetectable via timing measurement methods.

2 Related Work

Deception-based information resources that have no production value other than to attract and detect adversaries are commonly known as honeypots. Honeypots serve as effective tools for profiling attacker behavior and to gather intelligence to understand how attackers operate. They are considered to have low false positive rates since they are designed to capture only malicious attackers, except for perhaps an occasional mistake by innocent users. Spitzner discusses the use of honeytokens [13], which he defines as "a honeypot that is not a computer," citing examples that include bogus medical records, credit card numbers, and credentials. Our work harnesses the honeytoken concept to detect crimeware that may otherwise go undetected.

Injecting human input to detect malware has been shown to be useful by Borders *et al.* [14] with their Siren system. The aim of Siren is to thwart malware that attempts

to blend in with normal user activity to avoid anomaly detection systems. However, detection is performed by manually injecting human input to generate a sequence of network requests and observing the resulting network traffic to identify differences from the known sequences of requests; deviations are flagged as malicious. Expanding upon Siren, Chandrasekaran *et al.* [15], developed a system to randomize generated human input to foil potential analysis techniques that may be employed by malware. The work by Holz *et al.* [1] to investigate keyloggers and dropzones, relied on executing maleware in CWSandbox [16] and automating user input with AutoIt[3]. However, it was limited to ad hoc scenarios designed for the sole purpose of detecting harvesting channels. Their approach depends on miss-configured and insecure dropzone servers to learn about what sort of information is being stolen. While this effort did reveal lots of interesting details about stolen information, it is limited by law and skill of the attackers (*i.e.,* they can just secure their dropzone servers). In addition, relying on simulator software that resides within the host, such as AutoIt, provides attackers with a simple means to detect and avoid it. In contrast to these systems, BotSwindler is difficult to detect, automatically injects input that is designed to be believable, relies on monitored decoy credentials for detection, and provides a platform to convince malware that it has captured legitimate credentials.

Taint analysis is another technique that has been used to detect credential stealing malware. Egele *et al.* [17] used taint analysis to track information as it is processed by the web browser and loaded in to browser helper objects (BHOs). Their approach allows for a human analyst to observe where information is being sent in offline analysis. Similarly, Yin *et al.* [18] built Panorama, a taint tracking system that extends beyond BHOs to handle tracking throughout multiple processes, memory swapping, and disks. These systems may work well to track information in a system, but they do so with large overhead (factor of 10-20 slowdown in the systems described) or contain components that reside on the guest [18]; both these features that can be detected by malware and used for evasion purposes.

BotSwindler injects monitored bait into VM-based hosts by simulating user activity that is of interest to crimeware. The simulation is performed on the native OS outside of the VM to minimize artifacts that could be used to tip-off resident malicious software. To keep track of the simulation state within the virtual environment, our approach relies on a form of virtual machine introspection (VMI), a concept proposed by Garfinkel and Rosenblum [19] to describe the act of inspecting a virtual machine's software from outside the virtual environment. The challenge of VMI lies in overcoming the semantic gap [20] between the two levels of abstraction represented by the VM and the underlying service or OS. Garfinkel and Rosenblum focused on inspecting memory, registers, device state, and other process related information to implement an attack resistant host-based IDS for VMs whereby the IDS is located outside of the guest in the virtual machine monitor (VMM). Other VMI implementations include [21,22,23], but unlike most of these approaches, we circumvent the semantic gap and rely on artifacts found in the VMM graphical framebuffer. To the best of our knowledge, we are the first to focus on the verification of state for user simulations, a challenge with unique requirements.

[3] `http://www.autoitscript.com`

3 BotSwindler Components

The BotSwindler architecture, as shown in Fig. 1, consists of two primary components including a simulator engine, VMSim, and a virtual machine verification component. Another aspect of BotSwindler (although not shown in the figure) are the monitored decoys that we employ for detecting malware. These components are described in the next three sections.

3.1 VMSim

BotSwindler's user simulator component, VMSim, performs simulations that are designed to convince malware residing inside the VM that command sequences are genuine. We posit that successfully creating a sequence of actions that tricks the malware into stealing and uploading a decoy credential can be achieved only if two essential requirements are met:

1. the simulator process remains undetected by the malware
2. the actions of the simulator appear to be generated by a human

We approach the first requirement by decoupling the location of where the simulation process is executed and where its actions are received. To do this, we run the simulator outside of a virtual machine and pass its actions to the guest host by utilizing the X-Window subsystem on the native host. The second requirement is addressed through a simulation creation process that entails recording, modifying, and replaying mouse and keyboard events captured from real users. To support this process, we leverage the Xorg Record and XTest extension libraries for recording and replaying X-Window events. The product is a simulator that runs on the native host producing human-like events without introducing technical artifacts that could be used to alert malware of the BotSwindler facade.

VMSim relies on formal language to specify the sequence of actions in the simulations. Representative details of the formal language are provided in Fig. 2 (many details

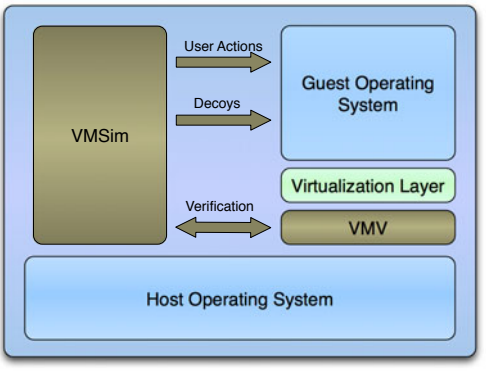


Fig. 1. BotSwindler architecture

are omitted due to space limitations). The language provides a flexible way to generate variable simulation behaviors and workflows, but more importantly it supports the use of *cover* and *carry* actions; carry actions result in the injection of decoys (described in Sect. 3.3), whereas cover actions include everything else to support the believability of carry traffic. For example, cover actions may include the opening and editing of a text document (`WordActions`) or the opening and closing of particular windows (`SysActions`). The `VerifyAction` allows VMSim to interact with VMV (described in Sect. 3.2) and provides support for conditional operations, synchronization, and error checking. Interaction with the VMV is crucial for the accuracy of simulations because a particular action may cause random delays for which the simulation must block on before proceeding to the next action.

```
<ActionType> ::= <WinLogin> <ActionType>
   | <CoverAction> <ActionType> | <CarryAction> <ActionType>
   | <WinLogout> | <VerifyAction> <ActionType> | e
<CoverAction> ::= <BrowserAction> <CoverAction>
   | <WordAction> <CoverAction> | <SysAction> <CoverAction>
<BrowserAction> ::= <URLRequest> <BrowserAction>
   | <OpenLink> <BrowserAction> | <Close>
<WordAction> ::= <NewDoc> <WordAction>
   | <EditDoc> <WordAction> | <Close>
<SysAction> ::= <OpenWindow> | <MaxWindow>
   | <MinWindow> | <CloseWindow>
<VerifyAction> ::= Img1 | Img2 | ... | ImgN | Unknown
<CarryAction> ::= <PayPalInject> | <GmailInject>
   | <CCInject> | <UnivInject> | <BankInject>
```

Fig. 2. VMSim language

The simulation creation process involves the capturing of mouse and keyboard events of a real user as distinct actions. The actions that are recorded map to the constructs of the VMSim language. Once the actions are implemented, the simulator is tuned to mimic a particular user by using various biometric models for keystroke speed, mouse speed, mouse distance, and the frequency of errors made during typing. These parameters function as controls over the language shown in Fig. 2 and aid in creating variability in the simulations. Depending on the particular simulation, other parameters such as URLs or other text that must be typed are then entered to adapt each action. VMSim translates the language's actions into lower level constructs consisting of keyboard and mouse functions, which are then outputted as X protocol level data that can be replayed via the XTest extensions.

To construct biometric models for individuals, we have extended QEMU's VMM to support the recording of several features including keycodes (the ASCII code representing a key), the duration for which they are pressed, keystroke error rates, mouse movement speed, and mouse movement distance. Generative models for keystroke timing are created by first dividing the recorded data for each keycode pair into separate classes where each class is determined by the distance in standard deviations from the mean. We then calculate the distribution for each keycode sequence as the number of instances of each class. We adapt simulation keystroke timing to profiles of individual users by generating random times that are bounded by the class distribution. Similarly, for mouse movements we calculate user specific profiles for speed and distance. Recorded mouse

movements are broken down into variable length vectors that represent periods of mouse activity. We then calculate distributions for each user using these vectors. The mouse movement distributions are used as parameters for tuning the simulator actions. We note that identifying the complete set of features to model an individual is an open problem. Our selection of these features is to illustrate a feasible approach to generating statistically similar actions. In addition, these features have been useful for verifying the identify of individuals in keystroke and mouse dynamics studies [24,25]. In Sect. 4.1 we provide a statistical and information theoretic analysis of the simulated times.

One of the advantages of using a language for the generation of simulation workflows is that it produces a specification that can be ported across different platforms. This allows the cost of producing various simulation workflows to be amortized over time. In the prototype version of BotSwindler, the task of mapping mouse and keyboard events to language actions is performed manually. The mappings of actions to lower level mouse and keyboard events are tied to particular host configurations. Although we have not implemented this for the prototype version of BotSwindler, the process of porting these mappings across hosts can be automated using techniques that rely on graphical artifacts like those used in the VMV implementation and applying geometric transformations to them.

Once the simulations are created, playing them back requires VMSim to have access to the display of the guest OS. During playback, VMSim automatically detects the position of the virtual machine window and adjusts the coordinates to reflect the changes. Although the prototype version of BotSwindler relies on the display to be open, it is possible to mitigate this requirement by using the X virtual frame buffer (Xvfb) [26]. By doing so, there would be no requirement to have a screen or input device.

3.2 Virtual Machine Verification

The primary challenge in creating an of out-of-host user simulator is to generate human-like events in the face of variable host responses. This task is essential for being able to tolerate and recover from unpredictable events caused by things like the fluctuations in network latency, OS performance issues, and changes to web content. Conventional in-host simulators have access to OS APIs that allow them to easily to determine such things. For example, simulations created with the popular tool AutoIt can call its `WinWait` function, which can use the Win32 API to obtain information on whether a window was successfully opened. In contrast, an out-of-host simulator has no such API readily available. Although the Xorg Record extensions do support synchronization to solve this sort of problem, they are not sufficient for this particular case. The Record extensions require synchronization on an X11 window as opposed to a window of the guest OS inside of an X11 window, which is the case for guest OS windows of a VM[4].

We address this requirement by casting it as a verification problem to decide whether the current VM state is in one of a predefined set of states. In this case, the states are defined from select regions of the VM graphical output, allowing states to consist of any visual artifact present in a simulation workflow. To support non-deterministic

[4] This was also a challenge when we tested under VMware Unity, which exports guest OS windows as what appear to be ordinary windows on the native host.

simulations, we note that each transition may end in one of several possible next states. We formalize the VMV process over the set of transitions T, and set of states S, where each $t_0, t_1, ..., t_n \in T$ can result in the the set of states $s_{t1}, s_{t2}, ..., s_{tn} \subseteq S$. The VMV decides a state verified for a current state c, when $c \in s_{ti}$.

The choice for relying on the graphical output allows the simulator to depend on the same graphical features a user would see and respond to, enabling more accurate simulations. In addition, information specific to a VM's graphical output can be obtained from outside of the guest without having to solve the semantic gap problem [20], which requires detailed knowledge of the underlying architecture. A benefit of our approach is that it can be ported across multiple VM platforms and guest OS's. In addition, we do not have to be concerned with side effects of hostile code exploiting a system and interfering with the Win32 API like traditional in-host simulators do, because we do not rely on it. In experiments with AutoIt scripts and in-host simulations, we encountered cases where scripts would fail as a result of the host being infected with malware.

The VMV was implemented by extending the Simple DirectMedia Layer (SDL) component of QEMU's [11] VMM. Specifically, we added a hook to the `sdl_update` function to call a VMV `monitor` function. This results in the VMV being invoked every time the VM's screen is refreshed. The choice of invoking the VMV only during `sdl_update` was both to reduce the performance costs and because it is precisely when there are updates to the screen that we seek to verify states (it is a good indicator of user activity).

States are defined during a simulation creation process using a pixel selection tool (activated by hotkeys) that we built into the VMM. The pixel selection tool allows the simulation creator to select any portion of a guest OS's screen for use as a state. In practice, the states should be defined for any event that may cause a simulation to delay (*e.g.,* network login, opening an application, navigating to a web page). The size of the screen selection is left up to the discretion of the simulation creator, but typically should be minimized as it may impact performance. In Sect. 4.3 we provide a performance analysis to aid in this consideration.

3.3 Trap-Based Decoys

Our trap-based decoys are detectable outside of a host by external monitors, so they do not require host monitoring nor do they suffer the performance burden characteristic of decoys that require constant internal monitoring (such as those used for taint analysis). They are made up of *bait information* including online banking logins provided by a collaborating financial institution, login accounts for online servers, and web based email accounts. For the experiments in this paper, we focused on the use of decoy Gmail, PayPal credentials, and banking credentials. These were chosen because they are widely used and known to have underground economy value [1,27], making them alluring targets for crimeware, yet inexpensive for us to create. The banking logins are provided to us by a collaborating financial institution. As part of the collaboration, we receive daily reports showing the IP addresses and timestamps for all accesses to the accounts at any time.

The decoy PayPal and bank accounts have an added bonus that allows us to expose the credentials without having to be concerned about an attacker changing their

password. PayPal requires multi-factor authentication to change the passwords on an account. Yet, we do not reveal all of the attributes of an account making it difficult for an attacker to change the authentication credentials. For the banking logins, we have the ability to manage the usernames and passwords.

Custom monitors for PayPal and Gmail accounts were developed to leverage internal features of the services that provide the time of last login, and in the case of Gmail accounts, the IP address of the last login. In the case of PayPal, the monitor logs into the decoy accounts every hour to check the PayPal recorded last login. If the delta between the times is greater than 75 seconds, the monitor triggers an alert for the account and notifies us by email. The 75 second threshold was chosen because PayPal reports the time to a resolution of minutes rather than seconds. The choice as to what time interval to use and how frequently to poll presents significant tradeoffs that we analyze in Sect. 4.4.

In the case of the Gmail accounts, custom scripts access `mail.google.com` to parse the bait account pages, gathering account activity information. The information includes the IP addresses for the previous 5 account accesses and the time. If there is any activity from IP addresses other than the BotSwindler monitor's host IP, an alert is triggered with the time and IP of the offending host. Alerts are also triggered when the monitor cannot login to the bait account. In this case, we conclude that the account password was stolen (unless monitoring resumes) and maliciously changed unless other corroborating information (like a network outage) can be used to convince otherwise.

4 Experimental Results

4.1 Statistical and Information Theoretic Analysis

In this section we present results from the statistical analysis of generated keystroke timing information. The goal of these experiments was to see if a machine learning algorithm (one that would be available to a malware sample to determine whether keystrokes are real or not) might be able to classify keystrokes accurately into user generated or machine generated. For these experiments, we relied on Killourhy and Maxion's benchmark data set [28]. The data set was created by having 51 subjects repeatedly type the same 10 character password, 50 times in 8 separate sessions, to create 400 samples for each user. Accurate timestamps were recorded by using an external clock. Using this publicly available real user data ensures that experiments can be repeated.

To evaluate VMSim's generated timing information, we used Weka [29] for our classification experiments. We divided the benchmark data set in half and used 200 password timing vectors from each user to train Naive Bayes and Support Vector Machine (SVM) classifiers. The remaining 200 timing vectors from each user were used as input to VMSim's generation process to generate 200 new timing vectors for each user. The same 200 samples were used for testing against the generated samples in the classification experiments. Note that we only used fields corresponding to hold times and interkey latencies because the rest were not applicable to this work (they can also contain negative values). The normalized results of running the SVM and Naive Bayes classifiers on the generated data and real data are presented in Figs. 3 and 4, respectively. The results are nearly identical for these two classifiers suggesting that this particular type

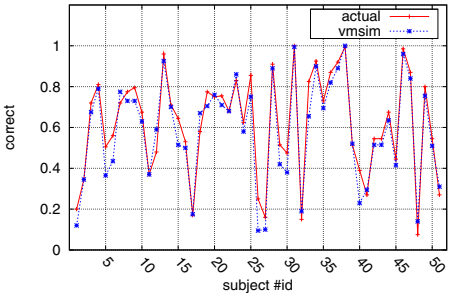


Fig. 3. SVM classification

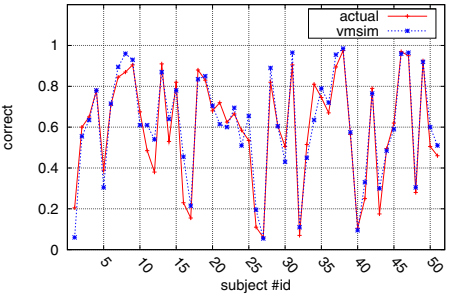


Fig. 4. Naive Bayes classification

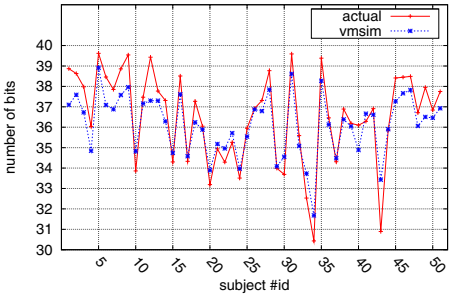


Fig. 5. Entropy of *generated* and *actual* timing data.

of analysis would not be useful for an attacker attempting to distinguish the real from generated actions. In Fig. 5, we present a comparison of entropy values (the amount of information or bits required to represent the data) [30] for the actual and generated data for each of the 200 timing vectors of the 51 test subjects. The results indicate that there is no loss of information in our generation process that would be useful by an adversary that is attempting distinguish real from generated actions.

4.2 Decoy Turing Test

We now discuss the results of a Turing Test [10] to demonstrate BotSwindler's performance regarding the *humanness*, or believability, of the generated simulations. The point of this experiments is to show that adversaries resorting to manual inspection of the user activities would be sufficiently challenged. Though the simulations are designed to delude crimware, here we focus on convincing humans, a task we posit to be a more difficult feat, making the adversaries task of designing malware that discerns decoys far more difficult. To conduct this study, we formed a pool of 25 human judges, consisting of security-minded PhDs, graduate-level students, and security professionals. Their task was to observe a set of 10 videos that capture typical user actions performed on a host and make a binary decision about each video: *real* or *simulated* (*i.e.,* whether the video shows the actions of a real user or those of a simulator). Our goal was to demonstrate the believability of the simulated actions by showing failure of

human judges to reliably distinguish between authentic human actions and those generated with BotSwindler. Our videos contained typical user actions performed on a host such as composing and sending an email message through Gmail, logging into a website of a financial institution such as Citibank or PayPal, and editing text document using Wordpad. For each scenario we generated two videos: one that captured the task performed by a human and another one that had the same task performed by BotSwindler. Each video was designed to be less than a minute long since we assumed that our judges would have limited patience and would not tolerate long-running simulations.

The human generated video samples were created by an independent user who was asked to perform sets of actions which were recorded with a desktop recording tool to obtain the video. Similar actions by another user were used to generate keystroke timing and error models, which could then be used by VMSim to generate keystroke sequences. To generate mouse movements, we rely on movements recorded from a real user. Using these, we experimentally determine upper and lower bounds for mouse movement speed and replay the movements from the real user, but with a new speed randomized within the determined limits. The keyboard and mouse sequences were merged with appropriate simulator parameters such as credentials and URLs to form the simulated sequence which was used to create the decoy videos.

Figure 6 summarizes the results for each of the 10 videos. The videos are grouped in per-scenario pairs in which the left bars correspond to simulated tasks, while the right bars correspond to the tasks of authentic users on which the simulations are based. The height of the bars reflects the number of judges that correctly identified the given task as real or simulated. The overall success rate was $\sim$46%, which indicates that VMSim achieves a good approximation of human behavior. The ideal success rate is 50%, which suggests that judges cannot differentiate whether a task is simulated or real.

Figure 7 illustrates the overall performance of each judge separately. The judges' correctness varies greatly from 0% up to 90%. This variability can be attributed to the fact that each judge interprets the same observed feature differently. For example, since VMSim uses real user actions as templates to drive the simulation, it is able to include advanced "humanized" actions inside simulations, such as errors in typing (*e.g.,* invalid typing of a URL that is subsequently corrected), `TAB` usage for navigating among form fields, auto-complete utilization, and so forth. However, the same action (*e.g.,* `TAB` usage for navigating inside the fields of a web form) is assumed by some judges as a real human indicator, while some others take it as a simulation artifact. This observation is clearly a "toss up" as a distinguishing feature. An important observation is that even highly successful judges could not achieve a 100% accuracy rate. This indicates that given a diverse and plentiful supply of decoys, our system will be believable at some time. In other words, given enough decoys, BotSwindler will eventually force the malware to reveal itself. We note that there is a "bias" towards the successful identification of bogus videos compared to real videos. This might be due to the fact that most of the judges guess "simulated" when unsure, due to the nature of the experiment. Despite this bias, results indicate that simulations are highly believable by humans. In cases where they may not be, it is important to remember that the task of fooling humans is far harder than tricking malware, unless the adversary has solved the AI problem and designed malware to answer the Turing Test. Furthermore, if attackers have to

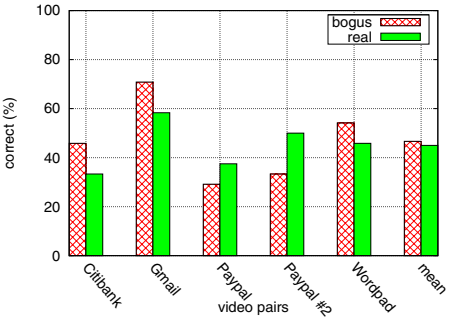


Fig. 6. Decoy Turing Test results: *real* vs. *simulated*

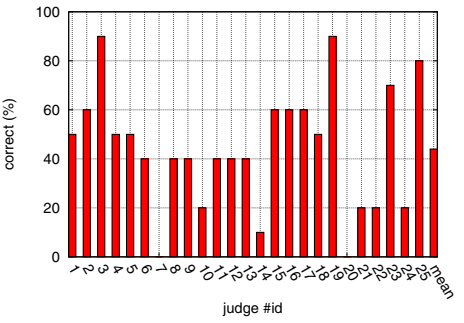


Fig. 7. Judges' overall performance

spend their time looking at the actions one by one to determine if they are real or not, we consider BotSwindler a success because that approach does not scale for the adversary.

4.3 Virtual Machine Verification Overhead

The overhead of the VMV in BotSwindler is controlled by several parameters including the number of pixels in the screen selections, the size of the search area for a selection, the number of possible states to verify at each point of time, and the number of pixels required to match for positive verification. A key observation responsible for maintaining low overhead is that the majority of the time, the VMV process results in a negative verification, which is typically obtained by inspecting a single pixel for each of the possible states to verify. The performance cost of this result is simply that of a few instructions to perform pixel comparisons. The worst case occurs when there is a complete match in which all pixels are compared (*i.e.,* all pixels up to some predefined threshold). This may result in thousands of instructions being executed (depending on the particular screen selection chosen by the simulation creator), but it only happens once during the verification of a particular state. It is possible to construct a scenario in which worse performance is obtained by choosing screen selections that are common (*e.g.,* found on the desktop) and almost completely matches but results in a negative VMV outcome. In this case, obtaining a negative VMV result may cost hundreds of thousands of CPU cycles. In practice, we have not found this scenario to occur; moreover, it can be avoided by the simulation creator.

In Table 1, we present the analysis of the overhead of QEMU[5] with the BotSwindler extensions. The table presents the amount of time, in seconds, to load web pages on our test machine (2.33GHz Intel Core 2 Duo with 2GB 667MHz DDR2 SDRAM) with idle user activity. The results include the time for a native OS, an unmodified version of QEMU (version 0.10.5) running Windows XP, and QEMU running Windows XP with the VMV processing a verification task (a particular state defined by thousands of pixels).

[5] QEMU does not support graphics acceleration, so all processing is performed by the CPU.

Table 1. Overhead of VMV with idle user

	Min.	Max.	Avg.	STD
Native OS	.48	.70	.56	.06
QEMU	.55	.95	.62	.07
QEMU w/VMV	.52	.77	.64	.07

Table 2. Overhead of VMV with active user

	Min.	Max.	Avg.	STD
Native OS	.50	.72	.56	.06
QEMU	.57	.96	.71	.07
QEMU w/VMV	.53	.89	.71	.06

In Table 2, we present the results from a second set of tests where we introduce rapid window movements forcing the screen to constantly be refreshed. By doing this, we ensure that the BotSwindler VMV functions are repeatedly called. The results indicate that the rapid movements do not impact the performance on the native OS, whereas in the case of QEMU they result in a $\sim$15% slowdown. This is likely because QEMU does not support graphics acceleration, so all processing is performed by the CPU. The time to load the web pages on QEMU with the VMV is essentially the same as without it. This is true whether the tests are done with or without user activity. Hence, we conclude that the performance overhead of the VMV is negligible.

4.4 PayPal Decoy Analysis

The PayPal monitor relies on the time differences recorded by the BotSwindler monitoring server and the PayPal service for a user's last login. The last login time displayed by the PayPal service is presented with a granularity of minutes. This imposes the constraint that we must allow for at least one minute of time between the PayPal monitor, which operates with a granularity of seconds, and the PayPal service times. In addition, we have observed that there are slight deviations between the times that can likely be attributed to time synchronization issues and latency in the PayPal login process. Hence, it is useful to add additional time to the threshold used for triggering alerts (we make it longer than the minimum resolution of one minute).

Another parameter that influences the detection rate is the frequency at which the monitor polls the PayPal service. Unfortunately, it is only possible to obtain the last login time from the PayPal service, so we are limited to detecting a single attack between polling intervals. Hence, the more frequent the polling, the greater the number of attacks on a single account that we can detect and the quicker an alert can be generated after an account has been exploited. However, the fact that we must allow for a minimum of one minute between the PayPal last login time and the BotSwindler monitor's, implies we must consider a significant tradeoff. The more frequent the polling, the greater the likelihood is for false negatives due to the one minute window. In particular, the likelihood of a false negative is:

$$P_{FN} = \frac{\text{length of window}}{\text{polling interval}}.$$

Table 3 provides examples of false negative likelihoods for different polling frequencies using a 75 second threshold. These rates assume only a single attack per polling interval. We rely on this threshold because we experimentally determined that it exhibits no false positives. For the experiments described in Sect. 4.5, we use the 1 hour polling frequency because we believe it provides an adequate balance (the false negative rate is relatively low and the alerts are generated quickly enough).

Table 3. PayPal decoy false negative likelihoods

Polling Frequency	False Negative Rate
.5 hour	.0417
1 hour	.0208
24 hour	.0009

4.5 Detecting Real Malware with Bait Exploitation

To demonstrate the efficacy of our approach, we conducted two experiments using BotSwindler against crimeware found in the wild. For the first experiment, we injected Gmail and PayPal decoys, and for second, we used decoy banking logins. The experiments relied on Zeus because it is the largest botnet in operation. Zeus is sold as a crimeware kit allowing malicious individuals to create and configure their own unique botnets. Hence, it functions as a payload dissemination framework with a large number of variants. Despite the abundant supply of Zeus variants, many are no longer functional because they require active command and control servers to effectively operate. This requirement gives Zeus a relatively short life span because these services become inactive (*e.g.,* they are on a compromised host that is discovered and sanitized). To obtain active Zeus variants, we subscribed to an active feed of binaries at the Swiss Security blog, which has a Zeus Tracker [6] and Offensive Computing[6].

In our first experiment, we used 5 PayPal decoys and 5 Gmail decoys. We deliberately limited the number of accounts to avoid upsetting the providers and having our access removed. After all, the use of these accounts as decoys requires us to continuously poll the servers for unauthorized logins as described in Sect. 4.4, which could become problematic with a large number of accounts. To further limit the load on the services, we limited the BotSwindler monitoring to once every hour.

We constructed a BotSwindler sandbox environment so that any access to `www.paypal.com` would be routed to a decoy website that replicates the look-and-feel of the true PayPal site. This was done for two reasons. First, if BotSwindler accessed the real PayPal site, it would be more difficult for the monitor to differentiate access by the simulator from an attacker, which could lead to false positives. More importantly, hosting a phony PayPal site enabled us to control attributes of the account (*e.g.,* balance and verified status) to make them more enticing to crimeware. We leveraged this ability to give each of our decoy accounts unique balances in the range of $4,000 - $20,000 USD, whereas in the true PayPal site, they have no balance. In the case of Gmail, the simulator logs directly into the real Gmail site, since it does not interfere with monitoring of the accounts (we can filter on IP) and there is no need to modify account attributes.

The decoy PayPal environment was setup by copying and slightly modifying the content from `www.paypal.com` to a restricted lab machine with internal access only. The BotSwindler host machine was configured with NAT rules to redirect any access directed to the real PayPal website to our test machine. The downside of using this setup is that we lack a certificate to the `www.paypal.com` domain signed by a trusted

[6] `http://www.offensivecomputing.net`

Certificate Authority. To mitigate the issue, we used a self-signed certificate that is installed as a trusted certificate on the guest. Although this is a potential distinguishing feature that can be used by malware to detect the environment, existing malware is unlikely to check for this. Hence, it remains a valid approach for demonstrating the use of decoys to detect malware in this proof of concept experiment. The banking logins used in the second experiment do not have this limitation, but they may not have the same broad appeal to attackers that make PayPal accounts so useful.

The experiments worked by automating the download and installation of individual malware samples using a remote network transfer. For each sample, BotSwindler conducted various simulations designed from the VMSim language to contain inject actions, as well as other cover actions. The simulator was run for approximately 20 minutes on each of the 116 binaries that were tested with the goal of determining whether attackers would take and exploit the bait credentials. Over the course of five days of monitoring, we received thirteen alerts from the PayPal monitor and one Gmail alert. We ended the study after five days because the results obtained during this period were enough to convince us the system worked[7]. The Gmail alert was for a Gmail decoy ID that was also associated with a decoy PayPal account; the Gmail username was also a PayPal username and both credentials were used in the same workflow (we associate multiple accounts to make a decoy identity more convincing). Given that we received an alert for the PayPal ID as well, it is likely both sets of credentials were stolen at the same time. Although the Gmail monitor does provide IP address information, we could not obtain it in this case. This particular alert was generated because Gmail detected suspicious activity on the account and locked it, so the intruder never got in.

We attribute the fewer Gmail alerts to the economics of the black market. Although Gmail accounts may have value for activities such as spamming, they can be purchased by the thousands for very little cost[8] and there are inexpensive tools that can be used to create them automatically. Hence, attackers have little incentive to build or purchase a malware mechanism, and to find a way to distribute it to many victims, only to net a bunch of relatively valueless Gmail accounts. On the other hand, high-balance verified PayPal accounts represent something of significant value to attackers. The 2008 Symantec Global Internet Security Threat Report [27] lists bank accounts as being worth $10-$1000 on the underground market, depending on balance.

For the PayPal alerts that were generated, we found that some alerts were triggered within an hour after the corresponding decoy was injected, where other alerts occurred days after. We believe this variability to be a consequence of attackers manually testing the decoys rather than testing through some automatic means. In regards to the quantity of alerts generated, there are several possible explanations that include:

- as a result of the one-to-many mapping between decoys and binaries, the decoys are exfiltrated to many different dropzones where they are then tested
- the decoy accounts are being sold and resold in the underground market where first the dropzone owner checks them, then resell them to others, who then resell them to others who check them

[7] We ended the study after 5 days, but a recent examination of the monitoring logs revealed alerts still being generated months after.

[8] We have found Gmail accounts being sold at $20 per 1000.

While the second case is conceivable for credentials of true value, our decoys lack any balance. Hence, we believe that once this fact is revealed to the attacker during the initial check, the attackers have no reason to keep the credentials or recheck them (lending support for the first case). We used only five PayPal accounts with a one-to-many mapping to binaries, making it impossible to know exactly which binary triggered the alert and which scenario actually occurred. We also note that the number of actual attacks may be greater than what was actually detected. The PayPal monitor polls only once per hour, so we do not know when there are multiple attacks in a single hour. Hence, the number of attacks we detected is a lower bound. In addition, despite our efforts to get active binaries, many were found to be inactive, some cause the system to fail, and some have objectives other than stealing credentials.

In the second experiment, we relied on several bank accounts containing balances over $1,000 USD. In contrast with the PayPal experiments, this experiment relied on an actual bank website with authentic SSL certificates. The bank account balances were frozen so that money could not actually be withdrawn. We ran the simulator for approximately 10 minutes on 59 new binaries. Over the course of five days of monitoring, we received 3 alerts from the collaborating financial institution. The point of these experiments is to show that decoy injection can be useful tool for detecting crimeware that can be difficult to detect through traditional means. These results validate the use of financial decoys for detecting crimeware. A BotSwindler system fully developed as a deployable product would naturally include many more decoys and a management system that would store information about which decoy was used and when it was exposed to the specific tested host.

5 Applications of BotSwindler in an Enterprise

Beyond the detection of malware using general decoys, BotSwindler is well suited for use in an enterprise environment where the primary goal is to monitor for site-specific credential misuse and to profile attackers targeting that specific environment. Since the types of credentials that are used within an enterprise are typically limited to business applications for specific job functions, rather than general purpose uses, it is feasible for BotSwindler to provide complete test coverage in this case. For example, typical corporate users have a single set of credentials for navigating their company intranet. Corporate decoy credentials could be used by BotSwindler in conducting simulations modeled after individuals within the corporation. These simulations may emulate system administrative account usage (*i.e.,* logging in as root), access to internal databases, editing of confidential documents, navigating the internal web, and other workflows that apply internally. Furthermore, software monocultures with similar configurations, such as those found in an enterprise, may simplify the task of making a single instance of BotSwindler operable across multiple hosts.

Within the enterprise environment, BotSwindler can run simulations on a user's system when it is idle (*e.g.,* during meetings, at night). Although virtual machines are common in enterprise environments, in cases where they are not used, they can be created on demand from a user's native environment. One possible application of BotSwindler is in deployment as an enterprise service that runs simulations over exported copies of

multiple users' disk images. In another approach, a user's machine state could be synchronized with the state of a BotSwindler enabled virtual machine [31]. In either case, BotSwindler can tackle the problem of malware performing long-term corporate reconnaissance. For example, malware might attempt to steal credentials only after they have been repeatably used in the past. This elevates the utility of BotSwindler from a general malware detector to one capable of detecting targeted espionage software.

The application of BotSwindler to an enterprise would require adaptation for site-specific things (*e.g,* internal URLs), but use of specialized decoys does not preclude the use of general decoys like those detailed in Sect. 3.3. General decoys can help the organization identify compromised internal users that could be, in turn, the target of blackmail, either with traditional means or through advanced malware [32].

6 Limitations and Future Work

Our approach of detecting malware relies on the use of deception to trick malware to capture decoy credentials. As part of this work, we evaluated the believability of the simulations, but we did so in a limited way. In particular, our study measured the believability of short video clips containing different user workflows. These types of workflows are adequate for the detection of existing threats using short-term deception, but for certain use cases (such as the enterprise service) it is necessary to consider long-term deception, and the believability of simulation command sequences over extended periods of time. For example, adversaries conducting long-term reconnaissance on a system may be able to discover some invariant behavior of BotSwindler that can be used to distinguish real actions from simulated actions, and thus avoid detection. To counter this threat, more advanced modeling is needed to be able to emulate users over extended periods of time, as well as a study that considers the variability of actions over time. For long-term deception, the types of decoys used must also be considered. For example, some malware may only accept as legitimate those credentials that it has seen several times in the past. We can have "sticky" decoy credentials of course, but that negates one of their benefits (determining when a leak happened).

Malware may also be able to distinguish BotSwindler from ordinary users by attempting to generate bogus system events that cause erratic system behavior. These can potentially negatively impact a simulation and cause the simulator to respond in ways a real user would not. In this case, the malware may be able to distinguish between authentic credentials and our monitored decoys. Fortunately, erratic events that result in workflow deviations or simulation failure are also detectable by BotSwindler because they result in a state that cannot be verified by the VMV. When BotSwindler detects such events, it signals the host is possibly infected. The downside of this strategy is that it may result in false positives. As part our future work we will investigate how to measure and manage this threat using other approaches that ameliorate this weakness.

7 Conclusion

BotSwindler is a bait injection system designed to delude and detect crimeware by forcing it to reveal itself during the exploitation of monitored decoy information. It relies on

an out-of-host software agent to drive user-like interactions in a virtual machine aimed at convincing malware residing within the guest OS that it has captured legitimate credentials. As part of this work we have demonstrated BotSwindler's utility in detecting malware by means of monitored financial bait that is stolen by real crimeware found in the wild and exploited by the adversaries that control that crimeware. In anticipation of malware seeking the ability to distinguish simulated actions from human actions, we designed our system to be difficult to detect by the underlying architecture and the believable actions it generates. We performed a computational analysis to show the statistical similarities of simulations to real actions conducted. To demonstrate the believability of the simulations by humans, we conducted a Turing Test that showed we could succeed in convincing humans about 46% of the time. Finally, Botwindler has been shown to be an effective and efficient automated tool to deceive and detect crimeware.

References

1. Holz, T., Engelberth, M., Freiling, F.: Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 1–18. Springer, Heidelberg (2009)
2. Stahlberg, M.: The Trojan Money Spinner. In: 17th Virus Bulletin International Conference (VB) (September 2007), http://www.f-secure.com/weblog/archives/VB2007_TheTrojanMoneySpinner.pdf
3. Researcher Uncovers Massive, Sophisticated Trojan Targeting Top Businesses. Darkreading (July 2009), http://www.darkreading.com/database_security/security/privacy/showArticle.jhtml?articleID=218800077
4. Higgins, K.J.: Up To 9 Percent Of Machines In An Enterprise Are Bot-Infected. Darkreading (September 2009), http://www.darkreading.com/insiderthreat/security/client/showArticle.jhtml?articleID=220200118
5. Song, Y., Locasto, M.E., Stavrou, A., Keromytis, A.D., Stolfo, S.J.: On the Infeasibility of Modeling Polymorphic Shellcode. In: 14th ACM Conference on Computer and Communications Security (CCS), pp. 541–551. ACM, New York (2007)
6. Blog, T.S.S.: ZeuS Tracker, https://zeustracker.abuse.ch/index.php
7. Messmer, E.: America's 10 most wanted botnets. Network World (July 2009), http://www.networkworld.com/news/2009/072209-botnets.html
8. Measuring the in-the-wild effectiveness of Antivirus against Zeus. Technical report, Trusteer (September 2009), http://www.trusteer.com/files/Zeus_and_Antivirus.pdf
9. Ilett, D.: Trojan attacks Microsoft's anti-spyware (February 2005), http://news.cnet.com/Trojan-attacks-Microsofts-anti-spyware/2100-7349_3-5569429.html
10. Turing, A.M.: Computing Machinery and Intelligence. Mind, New Series 59(236), 433–460 (1950)
11. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference, pp. 41–46. USENIX Association, Berkeley (April 2005)
12. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities. In: 11th Workshop on Hot Topics in Operating System (HotOS). USENIX Association, Berkeley (May 2007)

13. Spitzner, L.: Honeytokens: The Other Honeypot (July 2003), http://www.securityfocus.com/infocus/1713
14. Borders, K., Zhao, X., Prakash, A.: Siren: Catching Evasive Malware. In: IEEE Symposium on Security and Privacy (S&P), pp. 78–85. IEEE Computer Society, Washington (May 2006)
15. Chandrasekaran, M., Vidyaraman, S., Upadhyaya, S.: SpyCon: Emulating User Activities to Detect Evasive Spyware. In: Performance, Computing, and Communications Conference (IPCCC), pp. 502–509. IEEE Computer Society, Los Alamitos (May 2007)
16. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. In: IEEE Symposium on Security and Privacy (S&P), pp. 32–39. IEEE Computer Society, Washington (March 2007)
17. Egele, M., Kruegel, C., Kirda, E., Yin, H., Song, D.: Dynamic Spyware Analysis. In: USENIX Annual Technical Conference, pp. 233–246. USENIX Association, Berkeley (June 2007)
18. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In: 14th ACM Conference on Computer and Communications Security (CCS), pp. 116–127. ACM, New York (2007)
19. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: 10th Annual Network and Distributed System Security Symposium (NDSS). Internet Society, Reston (February 2003)
20. Chen, P.M., Noble, B.D.: When Virtual Is Better Than Real. In: 8th Workshop on Hot Topics in Operating System (HotOS), pp. 133–138. IEEE Computer Society, Washington (May 2001)
21. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: Tracking Processes in a Virtual Machine Environment. In: USENIX Annual Technical Conference, pp. 1–14. USENIX Association, Berkeley (March 2006)
22. Jiang, X., Wang, X.: "Out-of-the-Box" Monitoring of VM-Based High-Interaction Honeypots. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 198–218. Springer, Heidelberg (2007)
23. Srivastava, A., Giffin, J.: Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 39–58. Springer, Heidelberg (2008)
24. Monrose, F., Rubin, A.: Authentication via Keystroke Dynamics. In: 4th ACM Conference on Computer and Communications Security (CCS). ACM, New York (April 1997)
25. Ahmed, A.A.E., Traore, I.: A New Biometric Technology Based on Mouse Dynamics. IEEE Transactions on Dependable and Secure Computing (TDSC) 4(3), 165–179 (2007)
26. The XFree86 Project: XVFB(1), http://www.xfree86.org/4.0.1/Xvfb.1.html
27. Symantec: Trends for July - December 2007. White paper (April 2008)
28. Killourhy, K.S., Maxion, R.A.: Comparing Anomaly Detectors for Keystroke Dynamics. In: 39th Annual International Conference on Dependable Systems and Networks (DSN). IEEE Computer Society Press, Los Alamitos (June-July 2009)
29. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA Data Mining Software: An Update. ACM SIGKDD Explorations Newsletter 11(1), 10–18 (2009)
30. Lee, W., Xiang, D.: Information-Theoretic Measures for Anomaly Detection. In: IEEE Symposium on Security and Privacy (S&P), pp. 130–143. IEEE Computer Society, Washington (2001)
31. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 161–174. USENIX Association, Berkeley (April 2008)
32. Bond, M., Danezis, G.: A Pact with the Devil. In: New Security Paradigms Workshop (NSPW), pp. 77–82. ACM, New York (September 2006)

CANVuS: Context-Aware Network Vulnerability Scanning

Yunjing Xu, Michael Bailey, Eric Vander Weele, and Farnam Jahanian

Computer Science and Engineering, University of Michigan
2260 Hayward St., Ann Arbor, Michigan 48109, USA
{yunjing,mibailey,ericvw,farnam}@eecs.umich.edu

Abstract. Enterprise networks face a variety of threats including worms, viruses, and DDoS attacks. Development of effective defenses against these threats requires accurate inventories of network devices and the services they are running. Traditional vulnerability scanning systems meet these requirements by periodically probing target networks to discover hosts and the services they are running. This polling-based model of vulnerability scanning suffers from two problems that limit its effectiveness—wasted network resources and detection latency that leads to stale data. We argue that these limitations stem primarily from the use of time as the scanning decision variable. To mitigate these problems, we instead advocate for an event-driven approach that decides when to scan based on changes in the *network context*—an instantaneous view of the host and network state. In this paper, we propose an architecture for building network context for enterprise security applications by using existing passive data sources and common network formats. Using this architecture, we built CANVuS, a context-aware network vulnerability scanning system that triggers scanning operations based on changes indicated by network activities. Experimental results show that this approach outperforms the existing models in timeliness and consumes much fewer network resources.

1 Introduction

Users in modern enterprise networks are assailed with spyware that snoops on their confidential information, spam that floods their e-mail accounts, and phishing scams that steal their identities. Network operators, whose goal is to protect these users and the enterprise's resources, make use of intrusion detection/prevention systems [22, 23], firewalls, and antivirus software to defend against these attacks. To be effective, the deployment and configuration of these systems require accurate information about the devices in the network and the services they are running. While both passive network-based and host-based methods for building these inventories exist, the most prevalent method of assessment continues to be active network-based vulnerability scanning. In this model, a small number of scanners enumerate the potential hosts in a network by applying a variety of tests to determine what applications and versions are being run and whether these services are vulnerable. For very large networks, scanning can take a significant amount of time (e.g., several weeks) and consume a large amount of network resources (e.g., Mbps). As a result, network operators frequently choose to run these scans only periodically.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 138–157, 2010.

Unfortunately, the dynamics of a hosts' mobility, availability and service configurations exacerbate the problem of when vulnerability scanning should take place. We define the knowledge of these changes as the context of the network. A context insensitive model for vulnerability scanning suffers from wasted resources (e.g., time, bandwidth, etc.) and the observation of stale data. For example, often the network operators who are responsible for protecting the network do not have full control over the placement and availability of hosts in the network. Addresses may be allocated to departments within the organization who use the addresses in different ways, leaving the network operators with little insight into what addresses are allocated or unallocated. Furthermore, these departments themselves often have little control over how their users make use of these resources and even known, allocated IP addresses and hosts may exhibit availability patterns that are difficult to predict. As a result, network operators spend resources and time scanning IP addresses that have not been allocated or for hosts that are unavailable.

In addition, network operators have limited visibility into what services are being run on these hosts because they are typically managed by different administrators. Without the knowledge about the context, the accuracy of detecting these services and their configurations is bound by the frequency of scanning. As a result, any change that occurs since the last scan will obviously not be visible until the next scanning iteration. The rapid occurrence of new, active exploits, announced vulnerabilities, and available software patches, along with the dynamic nature of how users utilize the network, suggest that even small drifts in these inventories may result in a large security impact for the organization. Furthermore, the assumption that services remain relatively static over a short period of time is increasingly flawed. The emergence of peer-to-peer, voice-over-ip, messaging, and entertainment applications have led to a large number of dynamic services on these hosts. Periodically scanning, by its very nature, only captures a snapshot of those services that are active at an instant in time and it may miss many other important services.

To solve these problems, we introduce a context-aware architecture that provides a uniform view of network states and their changes. The architecture makes use of existing sources of host behavior across a wide variety of network levels including the link, network, transport, and application layers. Diverse data formats such as syslog, SNMP, and Netflow representing activities at these layers are used to generate abstract views that represent important network activities (e.g., a host connecting to the network, a new subnet allocated, a new binary in use). Instead of scanning all the hosts in the network at the same frequency, periodic scanning in our architecture selectively scans hosts based on their availability patterns. Moreover, these abstract views are used to create events about host configuration changes (e.g., users connecting to a new service, downloads from update sites, and reboots) to trigger active scanning. Thus, this approach is inherently interrupt driven and this event-based model, on top of the context-aware architecture, provides more timely and accurate results. In contrast, scanning periodically at a higher frequency would be the alternative, but would require substantially more resources.

To demonstrate the effectiveness of this architecture, a prototype system is constructed and deployed in a large academic network consisting of several thousand active hosts distributed across a /16 and a /17 network. Evaluation of this architecture

over a 16-day period in March of 2010 illustrates that CANVuS outperforms existing techniques in detection latency and accuracy with a much fewer number of scans. The experimental results also reveal several problems of the current methodology including the lack of ground truth and the limited event types, both of which will be addressed in future work.

The rest of this paper is organized as follows: § 2 discusses research papers and commercial products that relate to enterprise network security, especially vulnerability assessment, and how our system differs from existing solutions. § 3 discusses our year-long evaluation of the university's scanning activities that lead to our current research. § 4 has an in-depth description of our context-aware architecture. Details of the CANVuS system implementation on this architecture is presented in § 5. § 6 describes the evaluation of CANVuS and the context-aware architecture. § 7 discusses the risks involved in this project and our mitigation efforts. The limitations and future work are explored in § 8. Finally, § 9 concludes the paper.

2 Related Work

A variety of security software solutions and appliances have been proposed to defend against the threats faced by enterprise networks. These fall roughly into those focused on real-time, reactive detection and prevention and those based on proactive risk identification and policy enforcement. Network-based, real-time detection and prevention solutions, such as intrusion detection systems [22, 23] are deployed at natural aggregation points in the network to detect or stop attacks buried in network packets by applying known signatures for malicious traffic, or by identifying abnormal network behaviors. Host-based antivirus software [32, 18] is meant to protect hosts from being infected by malicious programs before their binaries are executed and, like network-based approaches, may do so either through static signatures or anomaly detection.

In contrast, proactive approaches to network security seek to reason about risks before an attack event happens and to limit exposure to threats. To accomplish this form of proactive assessment and enforcement, these approaches require accurate views of the hosts, their locations, and the services running on them. One common way of determining this information is through the use of a network-based vulnerability scanner. Active network-based vulnerability scanners (e.g., Nessus [25], Retina [11]) operate by sending crafted packets to hosts to inventory the targets, providing fingerprints of the host operating systems and the host network services. Conversely, passive scanners [26, 31, 8, 17] fingerprint software versions by auditing their network traffic and matching them with the signature database. They can continuously monitor target networks and are less intrusive to the targets. However, their scope is limited by the traffic they have access to and, as a result, passive scanners are usually deployed alongside active scanners. In addition to these generic scanners, there has been a great deal of recent work in specialized scanners that evaluate the security of popular applications such as web applications [15, 6].

Once the accurate inventory and service data is acquired, it can be used for a variety of tasks. For example, firewalls [9] are available to both networks and end hosts to enforce administrator polices, to block unwanted services [1, 2], and to prioritize the

patching of vulnerable services [19, 7] before they are exploited. Often this reasoning makes use of attack graph representations of this inventory and service data to make their placement and configuration services. An attack graph is a graphical representation of all possible ways to compromise hosts in a network by taking advantage of its vulnerabilities. Sheyner et al. did the early work of attack graph generation using a model checking approach [27]. Subsequently, several improvements [5, 21, 20, 13] have been proposed to solve the scalability problem of the original attack graph approach. Another improvement is the introduction of link analysis methods in attack graphs to automate the analysis process [16, 24].

CANVuS varies from this existing work in that it does not provide new active or passive tests to determine a host configuration, nor does it propose a new representation or application of the host and service inventory data. Rather, the proposed architecture seeks to provide more up-to-date data with fewer costs than existing approaches by leveraging network context. In this sense, our work is relevant to other work in utilizing context to improve the performance and accuracy of a variety of security techniques [29]. For example, Sinha et al. leveraged the characteristics of the network workload to improve the performance of IDSes [30] and showed that building honeypots that are consistent with the network could improve the resilience of honeypots to attacks and improve their visibility [28]. Notions of managing numerous remote probing devices through a middleware layer was explored, though only in the context of IDSes, in the Broccoli system [14]. Cooke et al. built the Dark Oracle [10] that closely resembles the work in this paper in terms of methodology by using context-aware information to provide a database of network state, but it addressed primarily allocation information. Allman et al. proposed a general framework that also uses a trigger-based approach to do reactive network measurement [4]. While this is similar to our work in terms of the high-level idea, it tries to solve a different problem, and it contains no implementation or evaluation to demonstrate the effectiveness of their approach. More generally, to address the problems of comprehensive network visibility, a set of guidelines were outlined in [3] for three broad categories — basic functionality, handling and storage of data, and crucial capabilities. To our knowledge, no work has fully addressed all of these guidelines, although some work has been attempted to address the storage and querying of this ubiquitous visibility over time and space [34]. Our work makes progress towards comprehensive network visibility with the goal of creating a flexible, yet efficient unified network visibility system for CANVuS.

3 Motivation

The motivation for this work derives directly from our interactions with the University of Michigan's office of Information and Infrastructure Assurance (IIA) [33]. This group is tasked with: "(i) Facilitating campus-wide incident response activities, (ii) Providing services such as security assessments and consultation, network scans, education and training, and (iii) Managing IT security issues at the university level." As part of these roles, this office engages in quarterly scans of seven /16 subnets belonging to the University of Michigan. As part of an effort to evaluate this process, we assisted the IIA staff in analyzing several quarterly scans of this space by using both Nessus [25] and

Retina [11]. The results of this analysis were kept private to assist the security operation staff, but we were struck by several poignant observations from the experience:

- The scans generally take one and a half to two weeks to complete.
- In an effort to reduce the amount of time spent scanning, a significant number of vulnerability signatures present in the tools were not used.
- With the exception of a handful of departments, the scans of the IP space proceeded without knowledge of sub allocations in each department, scanning large blocks of space in their entirety.
- Due to the impact of work day availability patterns, the operators schedule the scans to occur only during working hours (i.e., 8 AM to 5 PM, Monday through Friday).
- Only 85% of the IP addresses in each scan were shared, the other 15% were unique.
- Only 85% of the total unique vulnerabilities discovered were present in both scans, with 15% of each scan's vulnerabilities appearing only in that scan.
- Only 56% of the configurations between two scans were unchanged for those IPs in common between the scans.

While surprising to us, the IIA staff were keenly aware of the dynamic nature of their network and the overhead imposed by the scanning activities. Although they deployed several stop-gap measures to deal with the effect of this dynamic network context (e.g., scan during work hours), these operators simply lacked the platform with which to achieve network-wide visibility.

4 Architecture and Design

In this section, we describe a context-aware architecture that provides a uniform view of network states and their changes for security applications. The architecture consists of three major components. The first component is a set of network monitors that are distributed over many network devices. The list of network devices to monitor could include switches, routers, and servers, but the architecture allows for other similar devices as well. The second major architectural component is a Context Manager, which converts data from network monitors to a network state database. The third and final component is the network state database that provides a uniform model for context-aware vulnerability scanning. Other context-aware applications may be built upon this database as well. A high-level diagram with the major components of the architecture is illustrated in Figure 1.

The design of this context-aware architecture is informed strongly by the design principles outlined in Allman et al. [3], especially those basic guidelines of scope, incremental deployability, and operational realities. We aim for a system that built for an individual enterprise and utilizes existing sources of data collected from infrastructure and services already deployed in the network. We utilize the existing common data formats (e.g., syslog, SNMP, Netflow) and store and access this data through common, extensible mechanisms (e.g., databases, SQL). Where necessary, we support probe-based mechanisms for extracting similar data from network data streams in the event that existing hardware is overloaded or does not support data export. With respect to the outlined data-oriented goals, we opt to focus on exploring data breadth over

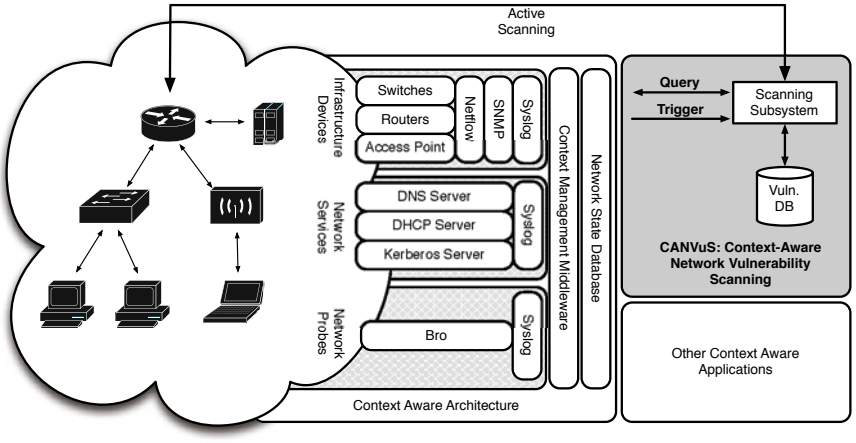


Fig. 1. Our context-aware network vulnerability scanning (CANVuS) architecture. The enterprise network is monitored by using data from existing physical infrastructure devices, network service appliances, and generic network probes. These heterogeneous sources of data are combined into a unified view of the network context which can be queried by context-aware applications or can have triggers automatically executed in response to certain contextual changes. In the case of CANVuS, contextual changes that indicate possible configuration changes are used to more efficiently scan network devices.

long term storage, smart storage management, graceful degradation, etc. The CANVuS application does not require extensive historical data, although we acknowledge that other context-aware applications will indeed require these functionalities and we look to leverage existing work in this direction for future versions of our architecture [34].

In the next three subsections, we first describe categories of monitoring points or data sources, from which creates a view of network context. Using this understanding, we then present the design of the Context Manager, which converts data from network monitors to the network state database in a uniform representation. Finally, we provide an example of what the network state database would look like for context-aware applications.

4.1 Sources of Data

Inferring network states and state changes is a challenging problem because the required information is distributed across many devices, network services, and applications. Thus, the key to capture the states and changes is to monitor the targets from a network perspective and approximate the context by aggregating network events from various data sources, which may lay in different layers of the network stack. To determine what data sources to use for event collection and integration, we first need to understand what types of network activities could be monitored and how they relate to changes in the network.

In this architecture, the monitors distributed across the network fall into three categories. The first category is monitors deployed in infrastructure devices, such as switches, routers and wireless access points. The advantage of having these types of monitors is that they provide detailed knowledge about the entire network, as well as the host information, with high resolution. This is a direct result of these devices providing the core hardware infrastructure for the network. For example, by querying the switches with SNMP, the system knows exactly when a particular physical machine is plugged into the network and when it leaves. However, because of the importance of these devices, tradeoffs are involved between the fidelity of data and the overhead (or difficulty) of the data collection.

The second category of monitors is for network services, which may include DNS, DHCP and even Kerberos services. The data observed by this second category of monitors provides additional detailed states about the hosts. Services in this class are the ones that are deployed in almost every enterprise and are critical to the operation of the network. For example, the data generated for DHCP service helps to distinguish between configuration changes for the same host and MAC-to-IP binding changes for different hosts. Depending on the types of the network, monitors in this category may require access to the syslog that are local to each service.

The final category consists of passive probes, which are deployed along with packet taps. These probes perform realtime analysis of network traffic that can be either generic TCP/UDP flows or application/OS specific to generate events. By monitoring TCP/UDP flows, the system gains the knowledge of which hosts are available and what services are running and being used by clients. In addition, using Deep Packet Inspecting (DPI) for an application protocol or OS fingerprint helps inventory the specific versions of applications (e.g., HTTP, SSH) or operating systems running in the network. Deviating version information for the same host directly indicates configuration changes for that host. For example, one could use Bro's protocol analysis to do connection reconstruction and application version fingerprinting [22] (which we used in our implementation of CANVuS). However, the visibility of passive probes is limited to the network traffic they have access to. In other words, silent applications/services will be missed, which is also the limitation of passive vulnerability scanners [26, 8, 17]. Thus, the passive probe is merely an additional type of monitor for collecting context data for observing a complete view of the network.

We acknowledge that host-based monitors exist for clients, but we decided that they are beyond the scope of this work. Intuitively, host-based monitors can provide the most accurate inventory information that effectively renders vulnerability scanning useless. Unfortunately, they are very difficult, if not impossible, to be enforced in large enterprise networks due to scalability and administrative reasons. However, as some traditional host-based programs, like antivirus software, are moved into the cloud [18], we may be able to build another type of monitors for potential in-cloud security services. The key advantage is that the in-cloud version of the services already have the visibility into the end hosts because they provide functionalities that used to be local to the hosts. Thus, the system will be able to obtain more fine-grained information about host activities without unacceptable modifications or performance penalties to the end hosts.

4.2 Context Manager

The Context Manager, a layer between data monitors and the network state database, infers network context (states and state changes) from aggregated data collected from various monitors and translates this to a uniform model. Specifically, the network states are a collection of simple facts about the target network, and they keep evolving as underlying hosts and services change. For example, these facts may include what hosts are available at the moment, what are the MAC addresses for a set of IPs in a certain period of time, or when is the first time a particular host connected to the network.

Existing libraries are used to read data from syslog, SNMP and Netflow that then get filtered and processed into a uniform representation of the network's state that is then inserted into the network state database. These modules can come from a programming languages standard library (e.g., Python's syslog module) or from third party libraries (e.g., PySNMP [12]). Additionally, the Context Manager supports a flexible framework for adding additional plugins to read input from new data monitoring sources and translate this data into a uniform model. These plugins can be thought of as data adapters that convert the source inputs canonical data format into a representation used in our architecture.

4.3 Network State Database

The network state database provides underlying model for which context-aware applications are written upon. The applications use standard database triggers or the programmable querying interfaces to interact with the database. The types of data that may be represented can be found in Table 1. This table shows how for that the data is uniformly represented across typical network abstraction layers, sources, and their respective data formats.

Table 1. Example contents of the network state database

Network Layer	Data Source	Data Format	Description
Link Layer	Switch	SNMP	Mapping between a MAC address and a Physical Switch Port
	Switch	SNMP	Mapping between a MAC address and an IP address
	Bro	Ethernet	Mapping between a MAC address and an IP address
Network Layer	Router	SNMP	Network allocation
Transport Layer	Router	Netflow	New connection established
	Bro	TCP/IP	New connection established
Application Layer	DNS Server	Syslog	Name resolution
	Bro	UDP/IP	Name resolution
	DHCP Server	Syslog	Mapping between a MAC address and an IP address
	Bro	UDP/IP	Mapping between a MAC address and an IP address
	Bro	TCP/IP	Application version fingerprint

5 CANVuS

In this section, we describe the implementation of CANVuS, a vulnerability scanning system, based on our context-aware architecture. It was implemented in Python to connect the network state database and a vulnerability scanner. In our implementation, we

used Nessus [25]. Ideally, if all host configuration changes produce network artifacts, the need for network vulnerability scanning of these events would be straightforward. However, not all host changes have network evidence that can be captured by at least one of the monitors. Thus, CANVuS uses both query and callback interfaces from the network state database to leverage the context information and to maintain a history of scanning results in its own vulnerability database.

During the initialization phase, CANVuS queries the database for all available hosts as scanning candidates. Due to the constraints in hardware and network resources, all hosts are not scanned concurrently. Instead, candidates are queued for pending scans, whose size depends on the network conditions, the configured policies, and the amount of available physical resources. A scheduling strategy is needed here to select the next candidate to scan. For example, each candidate could be weighted based on their triggering events and scheduled accordingly. In the current implementation, a simple FIFO strategy is applied. At the same time, CANVuS registers a callback function with database triggers so that a new candidate will be appended to the pending queue when a change happens, unless the same target is already in the queue. To conduct actual scanning operations, Nessus is used, yet is modified to change its target pool from a static configuration file to the queue discussed above.

Conversely, if a scanned host has no events fired after N seconds since its last scan, and there is no evidence (including both the context information and former scanning results) indicating that it becomes unavailable, it will be added to the queue for another scan. Thus, each host is effectively equipped with its own timer. Once it expires, CANVuS will query the network state database and its vulnerability database to determine if further scanning is necessary. Clearly, the value of the timer is a configurable policy up to the decision of the operators. In addition, when registering callback functions, instead of simply subscribing all possible changes in the database, CANVuS defines a set of event-specific policies to filter the ones that are less relevant to host configuration changes.

The choices of polices involve tradeoffs and depend on the objectives of the administrators who manage this system. The purpose of our current implementation is not to provide a reference system for production use. Instead, we aim to figure out what events are more effective in detecting changes and what policies are more appropriate with the given network conditions and administrative requirements. Therefore, the policies used for our experiment were set to be as aggressive as possible so that an optimal solution could be determined by filtering unnecessary scans after the experiment was complete. More specifically, the default policy is that every single events triggers a scan. The only exception is the TCP event, since there are too many of them for each host, an active timeout is enforced to prevent them from overwhelming the system. On the other hand, if scans are being constantly triggered by inbound connections to ports that scanners fail to discover, a negative timeout is also enforced to suppress this type of event being fired over and over again. Further details regarding the revision of our trigger implementation and policy decisions are presented in the evaluation.

The vulnerability database is the central storage of vulnerability data for all of the hosts in the network. It keeps track of the result for every scan conducted against each

host. In addition to the raw results generated by our modified Nessus scanner, each scan record contains following information:

- The time when this scan is triggered.
- The time when the backend scanner starts and finishes the scan.
- A list of open ports and the vulnerabilities on each port.
- A map from open ports to services.
- Operating system fingerprint (optional).
- The type of triggering event.

As more results are inserted into the vulnerability database, the information can be used in policy evaluation for further scans. Additionally, this information may be queried by administrators at any time for risk assessment or used by other security applications.

6 Evaluation

In this section, we evaluate CANVuS in a large academic network. The basic metrics used throughout this section is the number of scans conducted, which represents the resource consumption or overhead, and the latency of detecting configuration changes, which represents the system efficacy. Ideally, CANVuS should outperform periodic scanning with fewer number of scans by implicitly avoiding examining unallocated IP addresses and unavailable hosts. Moreover, CANVuS should also achieve lower detection latency as many host configuration changes create network evidence that trigger scans timely in our architecture.

We begin by discussing our experimental methodology. Then we show how CANVuS outperforms existing models in terms of the number of scans required and the detection latency. Next, we explore the impact of timeout values to the CANVuS system. The following section evaluates the contribution of various data sources to CANVuS and their correlations with observed changes on the hosts. We conclude the evaluation by discussing the scalability requirements of the context-aware architecture.

6.1 Experimental Methodology

The target network for the experiment is a college-level academic network with one /16 and one /17 subnet. There are two measurement points for the experiments. One is the core router for the entire college network. Because it has the access to the traffic between the Internet and the college network, there were two monitors built on it:

- TCP connection monitor: records the creation of new TCP connections.
- Application version monitor: records the version strings in protocol headers.

The second measurement point is a departmental network within the college that has the visibility into both the inbound/outbound traffic and more fine-grained inter-switch traffic. As result, the following monitors were deployed:

- ARP monitor: records the ARP messages probing for newly assigned IPs.
- DHCP monitor: records DHCP acknowledgment events.

- DNS monitor: records queries to certain software update sites.
- TCP connection monitor: as described above.
- Application version monitor: as described above.

This choice of measurement points and event monitors enables CANVuS to cover the network stack from the link layer to the application layer. Moreover, it also allows us to analyze the effectiveness of individual monitors with different visibility.

Based on these event monitors, CANVuS was deployed to scan the college network using a 12-hour active/inactive timeout and 1-hour negative timeout. In addition, another instance of a Nessus scanner was deployed for comparison purposes. It was configured to constantly enumerate the entire college network (a.k.a. the loop scanner). Both scanners were restricted to allow a maximum of 256 concurrent scans. The experiment lasted for 16 days in March, 2010, during which time the loop scanner completed 46 iterations. And it was interrupted by a power outage for a couple of hours at the end of the first week. Since we expect the system to be running long term, occasional interrupts would not have a major impact to the experiment results.

We performed a direct comparison between CANVuS and the loop scanner across both dimensions of resource consumption and detection latency. During the current exceptionally aggressive experiment, the loop scanner took less then 9 hours to finish one iteration. In realistic deployments, we envision using larger values and scanning less aggressively. Thus, the result of the loop scanner is only considered to represent the best performance that traditional periodic scanning systems can achieve in detection latency. More realistic values are represented below by sampling multiple 9-hour periods.

Ground truth for the experiments was established by identifying the period in which an observable network change occurred. Specifically, the scanning records from both scanners are first grouped together based on the MAC (if available) or IP address of each target host, and then each group of records are sorted by the time when each scan started. In each sorted group, a change is defined as two consecutive scan records (scans of unavailable hosts are ignored) with different sets of open ports. In addition, we assume that the change always happens immediately after the former scan finishes. Subsequent discussions that require the knowledge about ground truth are all based on this model unless otherwise noted. We approximated the ground truth in this way because it is infeasible to gain the local access to a large number of hosts in the target network to collect the real ground truth. As a result, we will not be able to analyze the true positive rate of CANVuS, and the average latencies for both scanners represent the upper bound, or the worst case.

6.2 CANVuS Evaluation

Table 2 lists the number of scans conducted by CANVuS with a break down by event types and the total for the loop scanner. The loop scanner has an order of magnitude more scans than CANVuS because only about 20% of the IP addresses in the target network are known to be allocated, and at any instant of time, the number of available hosts are even less than that. However, the loop scanner has to exhaust the entire IP blocks unless the address allocation and host availability information can be statically encoded, which is rarely the case in enterprise networks [10].

Table 2. The numbers of scans conducted by CANVuS and the loop scanner

	CANVuS	Loop Scanner
Total	534,717	4,355,624
ARP	1.55%	N/A
DHCP	17.37%	
TCP	38.33%	
DNS	10.28%	
App. Protocol	0.03%	
Timeout	32.44%	

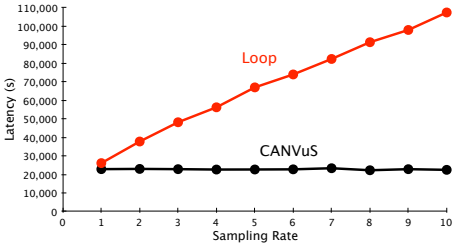


Fig. 2. A comparison of the detection latency with sampled results for the loop scanner

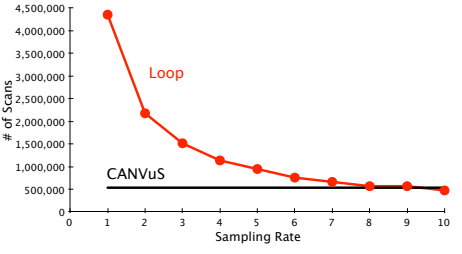


Fig. 3. A comparison of the number of scans with sampled results for the loop scanner

In addition, the average detection latencies for changes discovered by CANVuS is 22,868.751 seconds versus 26,124.391 seconds for the loop scanner. Please recall that our assumption says the evidence of configuration changes will trigger scans instantaneously. However, the latency for CANVuS here is far from zero. This anomaly is caused by the fact that we used the combined scanning results to approximate the ground truth and timeout-based scanning was still applied in some situations when no network network changes occurred.

Moreover, the latency for CANVuS is not significantly better than that of the loop scanner. This is because the configuration for the loop scanner is already very aggressive and represents the best performance that traditional periodic scanning systems can achieve in detection latency. To make the loop scans less aggressive and to demonstrate the tradeoff in scanning costs, the data set is sampled with a rate from 1 to 10 to include both the original case and the case that both scanners have a comparable number of scans. Figure 2 illustrates the result. The curve for the loop scanner goes up almost linearly because of the linear sampling, while the curve for CANVuS goes slightly up and down because the approximated ground truth has been changed after sampling. In addition, Figure 3 shows the corresponding changes for the number of scans.

6.3 Timeout-Based Scanning In CANVuS

As discussed previously, a timeout-based scanning approach is used along with the trigger-based scanning as many configuration changes are not observable through

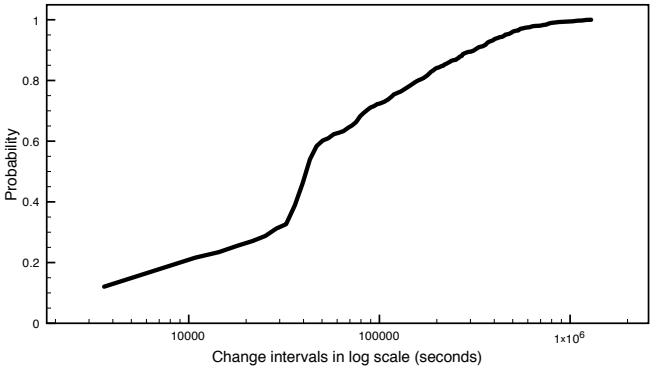


Fig. 4. The CDF for the intervals of detected changes

network events. However, unlike traditional periodic scanning, which randomly picks scanning targets in a large pool with fixed cycles, the timeout mechanism in CANVuS is still context-aware. Specifically, the system uses the context data to infer IP allocation information and host availability patterns so that only the hosts that are believed to be connected to the network will be scanned. These timeouts are assigned per-host timer and are based on the network state, scanning history, and administrators' policy decisions. As a result of these approach, fewer scans are "wasted" on hosts that don't exists or are unavailable. Taken another way, given the same number of scans, CANVuS is more likely to examine a larger number of active hosts and detect host changes with lower latency than the periodic scanning system.

Figure 4 depicts the distribution of intervals between *all* detected changes the values ranging between hours and days. The bias demonstrated in the middle of the graph is the result of our experimental methodology and the choice of 12 hours for our active/inactive timeout.

Thus, in practice, we determining the timeout value based on operators' administrative requirements. For example, using a timeout value at the order of a week, which covers the changes on the tail of the curve in Figure 4, would be reasonable. An alternative strategy is to halve the timeout value if a change is detected accordingly, but otherwise double it adaptively modifying a host's timeout value. In either case, there is clearly a tradeoff between the number of scans and the detection latency, which is also the case for normal periodic scanning.

6.4 Exploring the Impact of Various Data Sources and Triggers

In this subsection, we study the relationship between triggering events and the captured configuration changes. Specifically, the study is focused on their temporal correlations instead of causalities, which requires detailed control over the target hosts.

To do this, two problems need to be addressed. First, since most subnets in the target network use DHCP to assign IP addresses (despite whether the address mapping is dynamic or static), the changes witnessed may actually be mapping changes instead

Table 3. Permanent changes categorization

Triggering Event	Count	With network evidence
ARP	1	1
DHCP	15	10
TCP	2	2
DNS	3	1
Timeout	4	0

of changes in configuration. To eliminate the negative impact of dynamic address assignment, which would obscure the analysis results, the discussion in this subsection is confined to the 535 hosts that were assigned exactly one unique IP address during the experiment period. They are extracted from the 1,691 scanned IP addresses in the nine /24 departmental network subnets for which we have complete DHCP message logs.

Among these 535 hosts, 1,985 changes were detected by CANVuS during the experiment. After merging with the results from the loop scanner, the number of detected changes becomes 2,145, where the increase is an artifact of the method used to generate the approximated ground truth. However, many changes are considered to be ephemeral or short-lived, which is the second problem that must be handled. In other words, certain ports appear for a short while and then disappear without exhibiting real configuration changes. Many client programs may exhibit this behavior. For example, the client side of Apple's iTunes uses port 5353 to communicate with each other for sharing. P2P download software is another example. This type of changes provide little value in revealing the temporal correlations between changes and triggering events, due to their short lifetime.

As a result, we only consider those long-term or permanent changes to study their temporal correlations with triggering events. However, the word 'permanent' is not well defined, given the limited experiment period. Thus, for the simplicity of analysis, we only examine hosts that had exactly one change during the entire 16-day period because these changes are most likely to be permanent unless they were detected at the very end of the experiment. Among the 535 hosts, 25 of them fall into this category. Though this is not statistically significant, they still provide important clues for us to find appropriate policy setting for the target network. Recall that our conjecture is that most permanent changes have network evidence that can be witnessed and used for creating triggers to achieve timely detection. By manually going through all these changes and analyzing all logged events that happened around the time when the changes were detected, we find 56% of them have network evidence that has strong temporal correlation with changes, which means they either have triggered or could have triggered scans to detect the changes.

Table 3 is a summary of the analysis results in detail. Several things should be noted here. First, all the permanent changes we studied that were captured by timeout exhibited no network evidence at all. This is a limitation of our system using pure network events. Unless some level of host information is monitored, this timeout-based method cannot be completely replaced with the trigger-based approach. In addition, a significant portion of changes were detected via DHCP and ARP events, which corresponded

to hosts reboots. This is reasonable because many configuration changes may not take effect until the host is restarted. Finally, the rest of the changes corresponded to activities on the service or process level.

Moreover, we argue that although ephemeral changes may not be helpful in studying the temporal correlations, they are still relevant to risk assessment. Despite the possibility of being exploited by sophisticated attackers, many short-lived but well-known ports may be used to tunnel malicious traffic for cutting through firewalls. For example, the TCP event monitor captured some occasional events through port 80 for certain hosts, while the application event monitor failed to fingerprint any version information for them, which means the traffic did not follow the HTTP protocol. Thus, in both cases, it is valuable to detect these short-lived changes, but there is no guarantee for the loop scanner to achieve this goal. In fact, the loop scanner tends to miss most of these changes once its scanning period greatly increases (e.g., in the order of weeks). With the help of TCP events, CANVuS can fire scans immediately after there is traffic going through these ports. And there are 35 changes exclusively captured by CANVuS that fall in this category. Conversely, if there is no traffic ever going through the short-lived ports, while CANVuS may also miss them, the resulting risk is much lower because attackers have no chance to leverage them either.

6.5 Scalability Requirements of the Context-Aware Architecture

Figures 5 and 6 constitute a summary for the scale of the data from the departmental monitors done in hour intervals. The number of raw packets or flows per hour is counted in Figure 5. This raw data was observed at the various network monitors and probes before being converted by the Context Manager into the network state database. We note that the first three days worth of data are missing in our graph due to the misconfiguration of the monitoring infrastructure. For the duration of our experiment, we observed flows on the order of 16 million per hour at its peak and on average around 4 million per hour. Other noticeable observations include a typical average of 1 million DNS packets per hour and about 12 thousand ARP packets per hour. These four graphs in Figure 5 show that this system must handle adequately large volumes of traffic due to its distributed nature.

Figures 6 shows the number of events per hour that triggered scans after being converted by the Context Manager. Compared to the graphs in Figure 5, the volume of events generated from the raw data was greatly reduced. To highlight the number of flows and DNS packets went from the order of millions to the low hundreds. ARP packets when from the order of thousands to tens. This shows that our Context Manager is able to greatly reduce large volumes of data to something manageable for our event-based vulnerability scanning.

In addition, the cumulative number of unique MAC addresses in the departmental network is shown in Figure 7, which quantifies the scale of the physical boxes within the department (only for the second measurement point) that should be audited. We observed that slight bumps indicate new observances of MAC addresses over the course of a day while plateaus occurred over the weekends. We speculate that the observance of new, unique MAC addresses will level off if given a longer period of time to run our experiments. This graph also gives insight in bounding to the amount of raw and

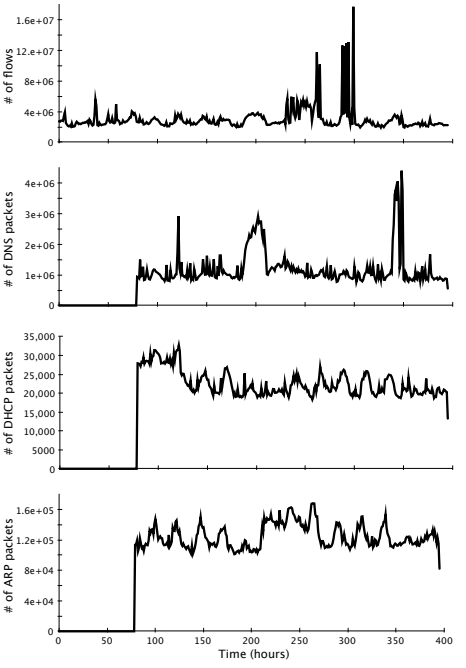


Fig. 5. The scale of the raw data

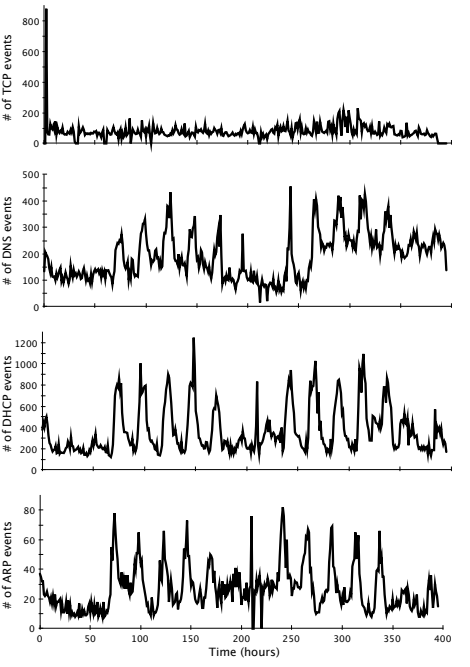


Fig. 6. The scale of the events

event-generated traffic that would be observed by the detection of fewer and fewer unique MAC addresses.

7 Risk Mitigation and Analysis

In this section, we examine our efforts in minimizing the harm to users, services, hosts, and network infrastructure while performing our experiments. We understand that active probing involves the use of network resources and interaction with product services. In consultation with the Computer Science and Engineering Departmental Computing Office, the College of Engineering Computer Added Engineering Network, and the University of Michigan office of Information and Infrastructure Assurance, we developed the following research plan to mitigate the impact on hosts, services, and network infrastructure: (i) to minimize the effect of network scanning, we limited the bandwidth available to our scanning devices, (ii) We implemented a whitelisting feature to our scanning, and the engineering computer organization broadcasted an opt-out message to all departmental organizations prior to our experiment (along with the complete research plan), (iii) We applied only those polices consistent with the Nessus "Safe Check Only" label.

Acknowledging that a network's security context is considered sensitive information and data such as MAC addresses and IP addresses have been viewed as personally

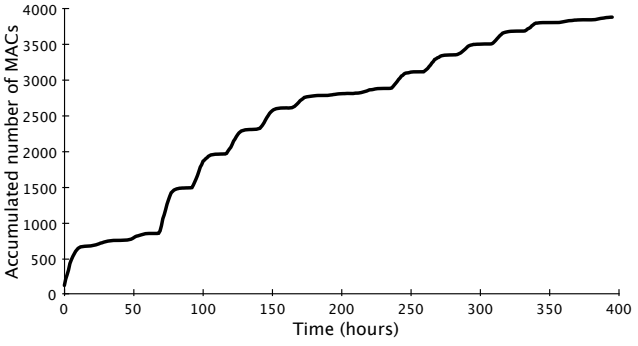


Fig. 7. The accumulated number of unique MAC addresses in the departmental network

identifiable information in several contexts, we took steps to assure that the "research records, data and/or specimens be protected against inappropriate use or disclosure, or malicious or accidental loss or destruction" according to our IRB guidelines. This includes, but is not limited to the following official categories: Locked office, Restricted access, Restrictions on copying study-related materials, Access rights terminated when authorized users leave the project or unit, Individual ID plus password protection, Encryption of digital data, Network restrictions, No non-UM devices are used to access project data, Security software (firewall, anti-virus, anti-intrusion) is installed and regularly updated on all servers, workstations, laptops, and other devices used in the project. Due to the technical nature of the work, we did not seek IRB approval for the project as we did not feel they were prepared to understand the risks of this work. The proposed research plan was instead approved through the College of Engineering Dean of Research, and additionally approved by the departmental, college, and university computing organizations specified above.

8 Limitations and Future Work

While our initial evaluation demonstrates the promise of a context-aware approach to vulnerability scanning, it does highlight several limitations which form the foundation for future work in this area. First, the accuracy of our evaluation is hampered by the use of network vulnerability scanning results as the sole ground truth for measuring host configuration changes. In addition to the previously discussed limitation that a network-based scanner provides only an approximate view of a host changes, this approach also limited the granularity of our measurements to the polling frequency of the network scanner. To overcome this issue, we plan on developing a host agent that is capable of collecting fine-grained information on local changes and deploying it on a network with a large number of different hosts (e.g., end hosts vs. application servers). A second rich area for future work is the exploration of new triggers (either new events or combinations of these events) for host configuration changes. Currently, the most effective events were generated by the DHCP monitor and corresponded to host reboots.

In the future, we plan to increase the diversity of trigger events and explore other types of network evidence for host changes.

9 Conclusion

In this paper, we proposed a context-aware architecture that provides information about the network states and their changes for enterprise security applications. We described how this architecture converts network data from infrastructure devices, network services, and passive probes to a uniform representation stored in the network state database. Then we introduced CANVuS, a context-aware vulnerability scanning system built upon this architecture that triggers scanning operations based on changes indicated by network activities. We experimentally evaluated the system by deploying it in a college-level academic network and comparing CANVuS against an existing system. We found that this approach outperforms existing models in low detection latency, while consuming fewer network resources.

Acknowledgments

The authors wish to gratefully acknowledge the following colleagues at the University of Michigan for their assistance in performing this work: Paul Howell, Kirk Soluk, Dawn Isabel, Dan Maletta, Kevin Cheek, and Donald Winsor. This work was supported in part by the Department of Homeland Security (DHS) under contract numbers NBCHC080037, NBCHC060090, and FA8750-08-2-0147, the National Science Foundation (NSF) under contract numbers CNS 091639, CNS 08311174, CNS 0627445, and CNS 0751116, and the Department of the Navy under contract N000.14-09-1-1042.

References

1. Abedin, M., Nessa, S., Al-Shaer, E., Khan, L.: Vulnerability analysis for evaluating quality of protection of security policies. In: Proceedings of the 2nd ACM Workshop on Quality of Protection (QoP 2006), Alexandria VA (October 2006)
2. Ahmed, M.S., Al-Shaer, E., Khan, L.: Towards autonomic risk-aware security configuration. In: Proceedings of the 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), Salvador, Bahia, Brazil (April 2008)
3. Allman, M., Kreibich, C., Paxson, V., Sommer, R., Weaver, N.: Principles for developing comprehensive network visibility. In: Provos, N. (ed.) Proceedings of 3rd USENIX Workshop on Hot Topics in Security, San Jose, CA, USA, July 29, USENIX Association (2008)
4. Allman, M., Paxson, V.: A reactive measurement framework. In: Claypool, M., Uhlig, S. (eds.) PAM 2008. LNCS, vol. 4979, pp. 92–101. Springer, Heidelberg (2008)
5. Ammann, P., Wijesekera, D., Kaushik, S.: Scalable, graph-based network vulnerability analysis. In: Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002), Washington DC (November 2002)
6. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: Automated black-box web application vulnerability testing. In: Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P 2010), Oakland, CA (May 2010)

7. Beattie, S., Arnold, S., Cowan, C., Wagle, P., Wright, C., Shostack, A.: Timing the application of security patches for optimal uptime. In: Proceedings of the 16th Annual LISA System Administration Conference, Philadelphia, PA, USA (November 2002)
8. Edward Bjarte. Prads - passive real-time asset detection system, http://gamelinux.github.com/prads
9. Cheswick, W.R., Bellovin, S.M.: Firewalls and Internet Security; Repelling the Wily Hacker. Addison Wesley, Reading (1994)
10. Cooke, E., Bailey, M., Jahanian, F., Mortier, R.: The dark oracle: Perspective-aware unused and unreachable address discovery. In: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 2006) (May 2006)
11. eEye Digital Security. Retina - network security scanner, http://www.eeye.com/Products/Retina.aspx
12. Ilya Etingof. Pysnmp, http://pysnmp.sourceforge.net/
13. Ingols, K., Lippmann, R., Piwowarski, K.: Practical attack graph generation for network defense. In: Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC 2006 (December 2006)
14. Kreibich, C., Sommer, R.: Policy-controlled event management for distributed intrusion detection. In: ICDCS Workshops, pp. 385–391. IEEE Computer Society, Los Alamitos (2005)
15. McAllister, S., Kirda, E., Kruegel, C.: Leveraging user interactions for in-depth testing of web applications. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 191–210. Springer, Heidelberg (2008)
16. Mehta, V., Bartzis, C., Zhu, H., Clarke, E., Wing, J.: Ranking attack graphs. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 127–144. Springer, Heidelberg (2006)
17. Microsoft. Watcher - web security testing tool and passive, http://websecuritytool.codeplex.com
18. Oberheide, J., Cooke, E., Jahanian, F.: Cloudav: N-version antivirus in the network cloud. In: Proceedings of the 17th USENIX Security Symposium (Security 2008), San Jose, CA (July 2008)
19. Oberheide, J., Cooke, E., Jahanian, F.: If It Ain't Broke, Don't Fix It: Challenges and New Directions for Inferring the Impact of Software Patches. In: 12th Workshop on Hot Topics in Operating Systems (HotOS XII), Monte Verita, Switzerland (May 2009)
20. Ou, X., Boyer, W.F., McQueen, M.A.: A scalable approach to attack graph generation. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006), Alexandria, VA (October 2006)
21. Ou, X., Govindavajhala, S., Appel, A.W.: Mulval: A logic-based network security analyzer. In: Proceedings of the 14th USENIX Security Symposium (USENIX Security 2005), Baltimore, MD (August 2005)
22. Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time. Computer Networks 31(23-24), 2435–2463 (1999)
23. Roesch, M.: Snort: Lightweight intrusion detection for networksx. In: Proceedings of the 13th Systems Administration Conference (LISA), pp. 229–238 (1999)
24. Sawilla, R.E., Ou, X.: Identifying critical attack assets in dependency attack graphs. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 18–34. Springer, Heidelberg (2008)
25. Tenable Network Security. Nessus - vulnerability scanner, http://www.nessus.org
26. Tenable Network Security. Nessus passive vulnerability scanner, http://www.nessus.org/products/pvs/
27. Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: Proceedings of 2002 IEEE Symposium on Security and Privacy (S&P 2002), Oakland, CA (May 2002)

28. Sinha, S., Bailey, M., Jahanian, F.: Shedding light on the configuration of dark addresses. In: Proceedings of Network and Distributed System Security Symposium (NDSS 2007) (February 2007)
29. Sinha, S., Bailey, M.D., Jahanian, F.: One Size Does Not Fit All: 10 Years of Applying Context Aware Security. In: Proceedings of the 2009 IEEE International Conference on Technologies for Homeland Security (HST 2009), Waltham, Massachusetts, USA (May 2009)
30. Sinha, S., Jahanian, F., Patel, J.M.: Wind: Workload-aware intrusion detection. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 290–310. Springer, Heidelberg (2006)
31. Sourcefire. Sourcefire rna - real-time network awareness, http://www.sourcefire.com/products/3D/rna
32. Sourcefire, Inc. Clamav antivirus (2008), http://www.clamav.net/
33. University of Michigan. University of Michigan — ITS — Safe Computing — IT Security Services Office (April 2010), http://safecomputing.umich.edu/about/
34. Vallentin, M.: VAST: Network Visibility Across Space and Time. Master's thesis, Technische Universitat Munchen (January 2009)

HyperCheck: A Hardware-Assisted Integrity Monitor

Jiang Wang, Angelos Stavrou, and Anup Ghosh

Center for Secure Information Systems
George Mason University, VA, USA
{jwanga,astavrou,aghosh1}@gmu.edu

Abstract. Over the past few years, virtualization has been employed to environments ranging from densely populated cloud computing clusters to home desktop computers. Security researchers embraced virtual machine monitors (VMMs) as a new mechanism to guarantee deep isolation of untrusted software components. Unfortunately, their widespread adoption promoted VMMs as a prime target for attackers. In this paper, we present HyperCheck, a hardware-assisted tampering detection framework designed to protect the integrity of VMMs and, for some classes of attacks, the underlying operating system (OS). HyperCheck leverages the CPU System Management Mode (SMM), present in x86 systems, to securely generate and transmit the full state of the protected machine to an external server. Using HyperCheck, we were able to ferret-out rootkits that targeted the integrity of both the Xen hypervisor and traditional OSes. Moreover, HyperCheck is robust against attacks that aim to disable or block its operation. Our experimental results show that HyperCheck can produce and communicate a scan of the state of the protected software in less than 40ms.

Keywords: Hypervisor, Protection framework, System Management Mode.

1 Introduction

Hypervisors[1] have become the de facto standard in server consolidation because they decrease the energy footprint and cost of management of modern computing clusters. In addition, hypervisors are increasingly used as components to enforce system security and resilience [22, 28, 16, 38, 21, 36, 31]. This widespread adoption of virtualization has attracted the attention of the attackers towards VMM vulnerabilities. Indeed, recently, there has been a surge in the reported vulnerabilities for commercial and open source hypervisors [27]. Moreover, the number and nature [40, 6] of attacks against the hypervisors are poised to grow.

This increasing attack trend has spurred research towards reducing the hypervisor Trusted Code Base (TCB) of current commercial hypervisors [26]. Others developed new specialized prototype hypervisors [36, 24]. However, having a small code base can only limit the code exposure and thus the attack surface of the hypervisor – it cannot provide strong guarantees about the code integrity of all the hypervisor components.

To address these limitations and to complement the existing protection mechanisms, we designed a hardware-assisted tampering detection framework called HyperCheck.

[1] Also called Virtual Machine Monitors VMMs.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 158–177, 2010.

HyperCheck is designed to protect the integrity of VMMs and, for some classes of attacks, the underlying operating system (OS). To achieve that, HyperCheck harnesses the CPU System Management Mode (SMM) which is present in all x86 commodity systems to create a snapshot view of the current state of the CPU and memory registers of the protected machine. This information is securely and verifiably transmitted using a network card to a remote analysis server. Using that information, the analysis server can identify any tampering by comparing the newly generated view with the one recorded when the machine was initialized. If the two views do not match, a human operator is notified. As shown in Figure 1, HyperCheck works at the BIOS level and can protect the software above it. Our assumptions are that the attacker does not have physical access to the machine and that the SMM BIOS is locked and thus cannot be altered during run. We do not explicitly require trusted boot to initialize HyperCheck [23, 24]. However, having a machine equipped with trusted boot can prevent attacks against HyperCheck that simulate a hardware reset.[2]

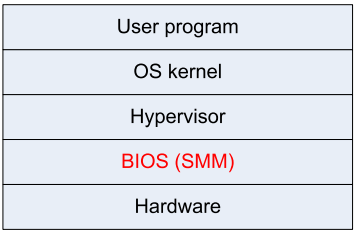


Fig. 1. HyperCheck can offer protection to services running above BIOS

Unlike previous work [30] that use specialized PCI hardware, we are able to acquire a complete view of the target machine's state including the entire memory and CPU registers. In addition, our approach is able to thwart attacks aimed at disabling, blocking, or even taking over PCI devices. To evaluate the validity and performance of our approach, we implemented two prototypes for HyperCheck. HyperCheck-I uses QEMU [3] – a fully system emulator – to emulate the PCI NIC, while HyperCheck-II is based on an Intel e1000 physical NIC. Using our prototypes, we were able to ferret-out rootkits aimed at Xen [11] hypervisor, Xen Domain 0, Linux, and Windows. Our experimental results indicate that HyperCheck does not cause prohibitive performance overhead requiring only a few milliseconds to completely transmit each snapshot.

In summary, we make the following contributions:

1. Designed a novel hardware-assisted tampering detection framework that creates a complete snapshot of the state of the system with commercial hardware and no modification to the installed software.
2. Implemented two prototypes: one based on QEMU and the other based on the real hardware. The latter has overhead in the order of few milliseconds. Using our prototype, we demonstrate that we can successfully detect rootkits and code integrity attacks against the Xen VMM, Xen Domain 0, Linux, and Windows.

2 Related Work

Protecting software from integrity attacks using hardware-assisted techniques is not new: researchers used a special-purpose PCI device to acquire the physical memory

[2] As we discuss in Section 7, the same can be accomplished using a management interface.

either for rootkit detection [30, 2] or for forensic purpose [8] in the past. The closest system to our work is Copilot [30]. Copilot employed a special PCI device to poll the physical memory of the host and send it to an admin station periodically. In HyperCheck, we do not require specialized hardware – only an out-of-the-box network card. We also offer a complete view of the CPU state including its registers. Such view is important to prevent copy-and-change attacks that can mislead the PCI card to scan the wrong regions of memory and report erroneously that the system is not affected.

Another closely related work is HyperGuard [33]. Rutkowska *et al.* suggested using SMM of the x86 CPU to monitor the integrity of the hypervisors. Although we have similar goals as the HyperGuard project, the use of a network card allows us to outsource the analysis of the state snapshot. This results in a drastic improvement in the performance of the system reducing the system busy time from seconds to milliseconds. Due to its low performance overhead, HyperCheck can also monitors the code and data of the privileged domain and underlying OSes. Another difference is that the monitoring machine can be used to detect the DoS attacks to the SMM code.

DeepWatch [6] also offers detection of hypervisor rootkits, called virtualization malware in DeepWatch, by using the embedded micro-controller(s) in the chipset. DeepWatch is signature based and used to detect rootkits relying on hardware-assisted virtualization technologies such as Intel VT-d [18]. Contrary, HyperCheck performs anomaly detection and thus can identify a larger class of software rootkits.

Flicker [23] uses a TPM based method to provide a minimum Trusted Code Base (TCB), which can be used to detect the modification to the kernels. Flicker requires advanced hardware features such as Dynamic Root of Trust Measurement (DRTM) and late launch. In contrast, HyperCheck uses the static Platform Configuration Registers (PCRs) to secure the booting process. In addition, by sending out the data, HyperCheck has a lower overhead on the target machine compared to Flicker. To reduce the overhead of Flicker, TrustVisor [24] has a small footprint hypervisor to perform some cryptography operations. However, all the legacy applications should be ported for TrustVisor to work. In addition, TrustVisor requires DRTM.

Another branch of research tries to improve the security of the hypervisor by adding hooks [10] and enforcing security policies between virtual machines [34]. These methods are hypervisor specific and run as the same level as the hypervisor. HyperCheck monitors the hypervisor state from a lower level and thus, is complementary to these methods.

Furthermore, there is a plethora of research aimed towards protecting the Linux kernel [2, 22, 16, 38, 21, 36, 31]. Baliga [2] *et al.* use a PCI device to acquire the memory and automatically derive the kernel invariance. Currently, we discover the kernel invariance manually but we could employ their techniques directly and without modifications. Litty [22] *et al.* developed a technique to discover the address of key data structures that are instantiated during run-time by relying on processor hardware and executable file specifications. But they also rely on the integrity of the underlying hypervisors. HyperCheck first obtains the virtual addresses of those symbols through the symbol file, but then calculates the physical addresses through CPU registers. Therefore, HyperCheck can get the correct view of the system memory even if the underlying OS or hypervisor is compromised and page tables are altered. Other existing research [38, 21, 36, 31],

including work by Jiang *et al.*, depend on the integrity of the hypervisor to protect the kernel. Our work is complementary and can be employed as a meta-protection mechanism to guard the integrity of OS-level defenses. A lot of recent work has gone towards using SMM to generate efficient rootkits [39, 5, 15, 12]. These rootkits can be used either to get root privilege or as a key-stroke loggers. We use SMM to offer integrity protection by monitoring the state of hypervisors and operating systems.

3 Threat Model

3.1 Background of System Management Mode

System Management Mode (SMM) was introduced in the Intel386 SL and Intel486 SL processors. It became a standard IA-32 feature in the Pentium [20] processor. SMM is a separate CPU mode besides the protected and real mode. The original purpose of SMM was to provide a transparent mechanism for implementing platform specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or a SMI is received from the advanced programmable interrupt controller (APIC) [20].

In SMM, the processor switches to a separate address space, called system management RAM (SMRAM). In addition, all hardware context of the currently running code is saved in SMRAM. Then, the CPU, being in SMM, executes transparently code that is usually a part of BIOS and resides in SMRAM. The SMRAM can be made inaccessible from other CPU operating modes. Therefore, it can act as trusted storage, sealed from being accessed from any device or even the CPU (while not in SMM mode). In HyperCheck, we modify the SMM code to execute our monitoring functions. This modification of SMM code can be integrated into the BIOS. Another way is to use a trust boot mechanism or a management interface to upload the code to SMM (when SMRAM is not locked) and then lock the SMRAM. Upon returning from SMM, the processor is placed back into its state prior to enter SMM.

3.2 Attacker's Capabilities

We assume that the adversary has following capabilities: she is able to exploit vulnerabilities in any software running in the machine after bootup. This includes the VMM and all of its privileged components. For instance, the attacker can compromise a guest domain and escape to the privileged domain. In Xen 3.0.3, pygrub [9] allows local users with elevated privileges in the guest domain (Domain U) to execute arbitrary commands in Domain 0 via a crafted grub.conf file [25]. Also, the attacker can modify the hypervisor code or data using any known or zero-day attacks. For instance, the DMA attack [40] hijacks a device driver to perform unauthorized DMA to the hypervisor's code or data.

3.3 General Assumptions

The attacker cannot tamper with, or replace the installed PCI NIC with a malicious NIC using the same driver interface. Also, if the SMM code is integrated with BIOS, we

assume the SMRAM is properly setup by BIOS upon boot time. If the SMM code is not included in the BIOS, it has to be reliably uploaded to the SMRAM during boot. This can be done by either using trusted boot or using the management interface to bootstrap the computer. In this case, to initialize the SMM code, a trusted bootstrap mechanism has to be employed. The SMRAM is locked once it is properly set up. Once it is locked, we assume it cannot be subverted by the attacker (an assumption supported by current hardware). Attacks that attempt to modify the SMM code [41, 13, 14] are beyond the scope of this paper.

3.4 In-Scope Attacks

HyperCheck aims to detect the in-memory, Ring-0 level (hypervisor or general OS) rootkits and rootkits in privileged domains of hypervisors. A rootkit is a set of programs and code that allows a permanent or consistent, undetectable presence on a computer [19]. One kind of rootkits only modifies the memory and/or registers and runs in the kernel level. For example, the idt-hook rootkit [1] modifies the interrupt descriptor table (IDT) in the memory and then gains the control of the complete system. An stealthier version of the idt-hook rootkit could keep the original IDT unchanged by copying it to a new location and altering it. Next, the attacker could change the IDTR register to point to the new location. When it comes to the hypervisor level rootkit, there is yet another kernel: the hypervisor kernel which runs underneath the operating system kernel. There are existing methods to detect in-memory, kernel-level rootkits. We try to bridge this gap by introducing HyperCheck.

3.5 Limitations

Currently, our analysis cannot protect against attacks that modify dynamic data. There are two types of threats: modification to the dynamically generated function pointers and return-oriented attacks. In these attacks, the control flow is redirected to memory location controlled by the attacker. There are techniques to thwart such attacks: the non-executable bit in new CPUs and Address Space Layout Randomization to name a few. HyperCheck can leverage and integrate those techniques to provide full protection but it was not part of our implementation in this paper. Having said that, we can still detect the presence of the malfease if it tries to interfere with the VMM code or statically defined function pointer.

4 System Architecture

HyperCheck is composed of three key components: the physical memory acquiring module, the analysis module and the CPU register checking module. The memory acquiring module reads the contents of the physical memory of the protected machine and sends them to the analysis module. Then, the analysis module checks the memory contents and verifies if anything is altered. The CPU register checking module reads

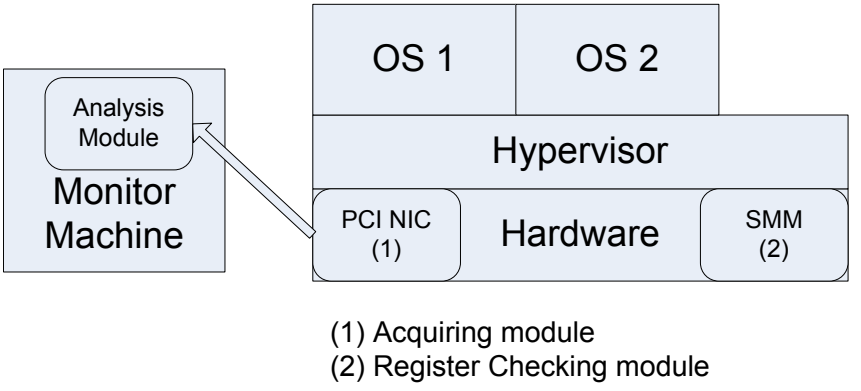


Fig. 2. The architecture of HyperCheck

the registers and validates their integrity. The overall architecture of HyperCheck is shown in Figure 2. Before introducing the key components, we first describe our design principles.

Our main design principle is that HyperCheck should not rely on any software running on the machine except the boot loader. Since the software may be compromised, one cannot trust even the hypervisor. Therefore, we use hardware – a PCI Ethernet card – as a memory acquiring module and SMM to read the CPU registers. Usually, Ethernet cards are PCI devices with bus master mode enabled and are able to read the physical memory through DMA, which does not need help from CPU. SMM is an independent operating mode and could be made inaccessible from protected mode which is what the hypervisor and privileged domains run in.

Previous researchers only used PCI devices to read the physical memory. However, CPU registers are also important because they define the location of active memory used by the hypervisor or an OS kernel such as CR3 and IDTR registers. Without these registers, the attacker can launch a copy-and-change attack. It means the attacker copies the memory to a new location and modifies it. Then the attacker updates the register to point to the new location. PCI devices cannot read the CPU registers, thereby failing to detect this kind of attacks. By using SMM, HyperCheck can examine the registers and report the suspicious modifications.

Furthermore, HyperCheck uses the CR3 register to translate the virtual addresses used by the kernel to the physical addresses captured by the analysis module. Since the acquiring module relies on the physical address to read the memory, HyperCheck needs to find the *physical* addresses of the protected hypervisor and privileged domain. For that purpose, HyperCheck checks both symbol files and CPU registers. From symbol files, HyperCheck can read the virtual addresses of the target memory. Then, HyperCheck utilizes CPU registers to find the physical addresses corresponding to the virtual ones. Previous systems only used the symbol files to read the virtual addresses and calculate the physical addresses. Such systems can not detect attacks that modify page tables and leave the original memory untouched. Another possible way to get the physical addresses without using registers, is to scan the entire physical memory and

use pattern matching to find all potential targets. However, this method is not scalable or even efficient especially since hypervisors and operating system kernels have small memory footprint.

4.1 Acquiring the Physical Memory

In general, there are two ways to acquire the physical memory: a software method and a hardware one. The former uses the interface provided by the OS or the hypervisor to access the physical memory, such as `/dev/kmem` on Linux [7] or \Device \PhysicalMemory on Windows [37]. This method relies on the integrity of the underlying operating system or the hypervisor. If the operating system or the hypervisor is compromised, the malware may provide a false view of the physical memory. Moreover, these interfaces to access memory can be disabled in future versions of the operating systems. In contrast, the hardware method uses a PCI device [8, 30] or other kinds of hardware [6]. The hardware method is more reliable because it depends less on the integrity of the operating system or the hypervisor.

We choose the hardware method to read the physical memory. There are also multiple options for the hardware components such as a PCI device, a FireWire bus device or customized chipset. We selected to use a PCI device because it is the most commonly used hardware. Moreover, existing commercial Ethernet cards need drivers to function. These drivers normally run in the operating system or the driver domain, which are vulnerable to the attacks and may be compromised in our threat model. To avoid this problem, HyperCheck puts these drivers into the SMM code. Since the SMRAM memory is going to be locked after booting, it will not be modified by the attacker. In addition, to prevent the attacker from using a malicious NIC driver in the OS to spoof the SMM driver, we use a secret key. The key is obtained from the monitor machine when the target machine is booting up and then stored in the SMRAM. The key then is used as a random seed to selectively hash a small portion of the data to avoid data replay attacks.

Another class of attacks is denial of service(DoS) attacks. Such attacks aim to stop or disable the device. For instance, according to ACPI [17] specification, every PCI device supports D3 state. This means that an ACPI-compatible device can be suspended by an attacker who takes over the operating system: ACPI was designed to allow the operating system to control the state of the devices. Of course, the OS is not a trusted component in our threat model. Therefore, one possible attack is to selectively stop the NIC without stopping any other hardware. To prevent ACPI DoS attacks, we need an out-of-band mechanism to verify that the PCI card is not disabled. The remote server that receives the state snapshots plays that role.

4.2 Translating the Physical Memory

In practice, there is a semantic gap between the physical memory that we monitor and the virtual memory addressing used by the hypervisor. To translate the physical memory, the analysis module must be aware of the semantics of the physical memory layout depends on the specific hypervisor we monitor. On the other hand, the acquiring module may support many different analysis modules with no or small modifications.

The current analysis module depends on three properties of the kernel memory: linear mapping, static nature and persistence. Linear mapping means the kernel (OS or hypervisor) memory is linearly mapped to physical memory and the physical addresses are fixed. For example, on x86 architecture, the virtual memory of Xen hypervisor is linearly mapped into the physical memory. Therefore, in order to translate the physical address to a given virtual address in Xen, we have to subtract the virtual address from an offset. In addition, Domain 0 of Xen is also linear mapped to the physical memory. The offset for Domain 0 is different on different machines but remains the same on a given machine. Moreover, other operating system kernels, such as Windows [35], Linux [4] or OpenBSD [12], also have this property when they are running directly on the real hardware.

Static nature means the contents of the monitoring part of the hypervisor have to be static. If the contents are changing, then there might be a time window between the CPU changing the contents and our acquiring module reading them. This may result in inconsistency for analysis and verification. Persistence property means the memory used by hypervisors will not be swapped out to the hard disk. If the memory is swapped out, then we cannot identify and match any content by only reading the physical memory. We would have to read the swap file on the hard disk.

The current version of HyperCheck relies on these three properties (linear mapping, static nature and persistence) to work correctly. Besides the Xen hypervisor, most operating systems hold these three properties too.

4.3 Reading and Verifying the CPU Registers

Since the Ethernet card cannot read the CPU registers, we must use another method to read them. Again, there are software and hardware based methods. For software method, one could install a kernel module in the hypervisor and then it could obtain registers by reading from the CPU directly. However, this is vulnerable to the rootkits, which can potentially modify the kernel module or replace it with a fake one. For hardware method, one could use a chipset to obtain registers.

We choose to use SMM in x86 CPU which is similar to a hardware method. As we mentioned earlier, SMM is a different CPU mode from the protected mode which the hypervisor or the operating system reside in. When CPU switches to SMM, it saves the register context in the SMRAM. The default SMRAM size is 64K Bytes beginning at a base physical address in physical memory called the SMBASE. The SMBASE default value following a hardware reset is 0x30000. The processor looks for the first instruction of the SMI handler at the address [SMBASE + 0x8000]. It stores the processor's state in the area from [SMBASE + 0xFE00] to [SMBASE + 0xFFFF] [20]. In SMM, if SMI handler issues `rsm` instruction, the processor will switch back to the previous mode (usually it is protected mode). In addition, the SMI handler can still access I/O devices. HyperCheck verifies the registers in SMM and reports the result by sending it via the Ethernet card to the monitor machine. HyperCheck focuses on monitoring two registers: IDTR and CR3. IDTR should never change after system initialization. For CR3, SMM code can use it to translate the physical addresses of the hypervisor kernel code and data. The offsets between physical addresses and virtual ones should never change as we discussed in Section 4.2.

5 Implementation

We implemented two prototypes for HyperCheck: HyperCheck-I is using QEMU full system emulation while HyperCheck-II is running on a physical machine. We first developed HyperCheck-I for quick prototyping and debugging. To measure the overall system performance, we implemented HyperCheck-II on non-virtualized hardware. Both of them utilize the Intel e1000 Ethernet card as the acquiring module.

In HyperCheck-I, the target machine is as a virtual machine that uses QEMU. The analysis module runs on the host operating system of QEMU. For the acquiring module, we placed a small NIC driver into the SMM of the target machine. Using the driver, we can program the NIC to transmit the contents of physical memory as an Ethernet frame. On the monitoring machine, an analysis module receives the packet from the network. The analysis module compares contents of the physical memory with the original (initial) versions. If a new snapshot of the memory contents is different from the original one, the module will report the event to a system operation who can decide how to proceed. Moreover, another small program runs in the SMM and collects and sends out the CPU registers also via the Ethernet card.

For HyperCheck-II, we used two physical machines: one as the target and the other as the monitor. On the target machine, we installed Xen 3.1 natively and used the physical Intel e1000 Ethernet card as the acquiring module. Also, we modified the default SMM code on the target machine to enable our system similarly to our QEMU implementation. The analysis module runs on the monitor machine and is the same as the one in HyperCheck-I. HyperCheck-II is mainly used for performance measurement.

As we mentioned earlier, we used QEMU for HyperCheck-I. QEMU is suitable for debugging potential implementation problems. However, it comes with two drawbacks. First, the throughput of a QEMU network card is much lower than a real NIC device. For our QEMU based prototype, the network card throughput is approximately 10MB/s, although Gigabit Ethernet cards are common in practice. Second, the performance measurement on QEMU may not reflect the real world performance. HyperCheck-II help us overcome these problems.

5.1 Memory Acquiring Module

The main task to implement the acquiring module is to port the e1000 network card driver into SMM to scan the memory and send it out. Normally, SMM code is one part of BIOS. Since we don't have the source code of the BIOS, we used the method similar to the one mentioned in [5] to modify the default SMM code. Basically, it writes the SMM code in 16bit assembly and uses a user level program to open the SMRAM and copy the assembly code to the SMRAM.

To overcome the limitations of [5], we divided the e1000 driver into two parts: initialization and data transfer. The initialization part is complex and very similar to the Linux driver. The communication part is simpler and different from the Linux driver. Therefore, we modified the existing Linux e1000 driver to initialize the network card and only program the transferring part in assembly. The e1000 driver on Linux is changed to only initialize the NIC but does not send out any packet. The assembly code is compiled to

an ELF object file. Next, we wrote a small loader which can parse the ELF object file and load the code and data to the SMM.

For this implementation, the NIC driver is ported to the SMM, the next step is to modify the driver to scan the memory and send them out. HyperCheck uses two transmission descriptors per packet, one for the header and the other for the data. The content of the header should be predefined. Since the NIC is already initialized by the OS, the driver in SMM has only to prepare the descriptor table and write it to the Transmit Descriptor Tail (TDT) register of the NIC. The NIC will send the packet to the monitoring machine using DMA. The NIC driver in SMM prepares the header data and let the descriptor point to this header. For the payload, the descriptor is directly pointed to the address of the memory that needs to be scanned. In addition, e1000 NIC supports CRC offloading.

To prevent replay attacks, a secret key is transferred from the monitor machine to the target machine upon booting. The key is used to create a random seed to selectively hash the data. If we hash the entire data stream, the performance impact may be high. To reduce the overhead, we use the secret key as a seed to generate one big random number used for one-time pad encryption and another set of serial random numbers. The serial of random numbers are used as the indexes of the positions of the memory being scanned. Then, the content at these positions are XORed with the one-time pad with the same length before starting NIC DMA. After the transmission is done, the memory content is XORed again to restore the original value.

The NIC driver also checks the loop-back setting of the device before sending the packet. To further guarantee the data integrity ,the SMM NIC driver stays in the SMM until all the packet is written to the internal FIFO of the NIC, and add 64KB more data to the end to flush the internal FIFO of the NIC. Therefore, the attacker cannot use loop-back mode to get the secret key or peek into the internal NIC buffer through debugging registers of the NIC.

5.2 Analysis Module

On the monitoring machine, a dedicated network card is connected with the acquiring module. The operating system of the monitoring machine was CentOS 5.3. We run `tcpdump` to filter the packets from the acquiring module; the output of `tcpdump` is sent to the analysis module. The analysis module written in a Perl script reads the input and checks for any anomalies. The analysis module first recovers the contents using the same secret key. After that, it compares every two consecutive memory snapshots bit by bit. If they are different, the analysis module outputs an alert on the console, as we are checking the persistent and static portion of the hypervisor memory. The administrator can then decide whether it is a normal update of the hypervisor or an intrusion. Note that during the system boot time, the contents of those control data and code are changing.

Currently, the analysis module can check the integrity of the control data and code. The control data includes IDT table, hypercall table and exception table of Xen, and the code is the code part of Xen hypervisor. To find out the physical address of these control tables, we use `Xen.map` symbol file. First, we find the virtual addresses of `idt_table`, `hypercall_table` and exception_table. The physical address of these symbols is virtual_address $-$ 0xff00,0000 on x86-32 architecture with PAE. The address of Xen hypervisor code is between `_stext` and `_etext`. HyperCheck can also

monitor the control data and codes of Domain 0. This includes the system call table and the code part of Domain 0 (a modified Linux 2.6.18 kernel). The kernel of Domain 0 is also linearly mapped to the physical memory. We use a kernel module running in Domain 0 to compute the exact offset. On our test machine, the offset is 0x83000000. Note that, there is no IDT table for Domain 0, because there is only one such table in the hypervisor. We input these parameters to the acquiring module to improve the scan efficiency.

Note that these control tables are critical to system integrity. If their contents are modified by any malware, it can potentially run arbitrary code in the hypervisor level, i.e. the most privileged level. An antivirus software or intrusion detection system that runs in Domain 0 is difficult or unable to detect this hypervisor level malware because they rely on the hypervisor to provide the correct information. If the hypervisor itself is compromised, it may provide fake information to hide the malware. The checking for the code part of the hypervisor enables HyperCheck to detect the attacks which do not modify the control table but just modify the code invoked by those tables.

5.3 CPU Register Checking Module

HyperCheck uses SMM code to acquire and verify CPU registers. In a product, the SMI handler should be integrated into BIOS. Or it can be set up during the system boot time. This requires the bootstrap to be protected by some trusted bootstrap mechanism. In addition, most chipsets provide a function to lock the SMRAM. Once it is locked, SMM handler cannot be changed until reboot. Therefore, the SMRAM should be locked once it is set up. In our prototype, we used the method mentioned in Section 5.1 to modify the default SMM code.

There are three steps for CPU register checking: 1) triggering SMI to enter SMM; 2) checking the registers in SMM; 3) reporting the result. SMI is a hardware interrupt and can only be triggered by hardware. Normally, I/O Controller Hub (ICH), also called Southbridge, defines the events to trigger SMI. For HyperCheck-I, the QEMU emulates Intel 82371SB chip as the Southbridge. It supports some device idle events to generate SMI. SMI is often used for power management, and Southbridge provides some timers to monitor the state of a device. If that device remains idle for a long time, it will trigger SMI to turn off that device. The resolutions of these timers are typically one second. However, on different motherboard, the method to generate the SMI may be different. Therefore, we employ the Ethernet card to trigger the SMI event.

For the register checking, HyperCheck monitors IDTR and CR3 registers. The contents of IDTR should never change after system boot. The SMM code just reads this register by `sidt` instruction. HyperCheck uses CR3 to find out the physical addresses of hypervisor kernel code and data given their virtual addresses. Essentially, it walks through all the page tables as a hardware Memory Management Unit (MMU) does. Note that offset between the virtual address and the physical address of hypervisor kernel code and data should never change. For example, it is 0xff000000 for Xen 32bit with PAE. The Domain 0 has the same property. The SMM code requires the virtual address range as the input (this can be obtained through the symbol file and send to the SMM in the boot time) and afterwards check their physical addresses. If any physical address is not

equal to virtual address – offset, this signifies a possible attack. The SMM code reports the result of this checking via the Ethernet card. The assembly code of it is just 67 LOC.

The SMM code uses the Ethernet card to report the result. Without the Ethernet card, it is difficult to send the report reliably without stopping the whole system. For example, the SMM code could write the result to a fixed address of physical memory. But according to our threat model, the attacker has access to that physical memory and can easily modify the result. Or the SMM code could write it to the hard disk. Again, this can be altered by the attacker too. Since security cannot relies on the obscurity, the only way left without a network card is to stay in the SMM mode and put the warning message on the screen. This is reliable, but the system in the protected mode becomes completely frozen. Sometimes, it may not be desirable, and could be abused by the attacker to launch Denial of Service attacks.

5.4 HyperCheck-II

In HyperCheck-II, the main difference from HyperCheck-I is the acquiring module. We ported the SMM NIC driver from QEMU to a physical machine. Both of them have the same model of the NIC: 82540EM Gigabit Ethernet card. However, the SMM NIC driver from the QEMU VM does not work on the physical machine. And it took one of the author one week to debug the problem. Finally, we find out that the main difference between a QEMU VM and the physical machine (Dell Optiplex GX 260) is that the NIC can access the SMRAM in a QEMU VM while it cannot on the physical machine. For HyperCheck-I SMM NIC driver, the TX descriptor is stored in the SMRAM and it works well. For HyperCheck-II, the NIC cannot read the TX descriptor in the SMRAM and therefore does not transmit any data.

To solve this problem, we reserved a portion of physical memory by adding a boot parameter: `mem=500M` to the Xen hypervisor or Linux kernel. Since the total physical memory on the physical machine is 512MB, we reserved 12MB for HyperCheck by using `mem` parameter. This 12MB is used to store the data used by SMM NIC and the TX descriptor ring. We also modified the loader to be a kernel module; it calls `ioremap()` to map the physical memory to a virtual address and load the data there. In a product, the TX descriptor ring should be prepared every time by the SMM code before transmitting the packet. In our prototype, since we don't have the source code of the BIOS, we used the loader to load the TX descriptor.

Finally, we built a debugging interface for the SMM code on the physical machine. We use the reserved physical memory to pass the information between the SMM code and the normal OS. This interface is also used to measure the performance of the SMM code as we will discuss in Section 6.

6 Evaluation

To validate the correct operation of HyperCheck, we first verified the properties that need to hold for us to be able to protect the underlying code as we discussed in Section 4.2. Then, we tested the detection for hypervisor rootkits and measured the operational overhead of our approach. We have worked on two testbeds: testbed 1 is mainly

used for HyperCheck-I and also as the monitor machine for HyperCheck-II. Testbed 2 uses HyperCheck-II to produce the plotted performance overhead on the real hardware. Testbed 1 was equipped with a Dell Precision 690 with 8GB RAM and one 3.0GHz Intel Xeon CPU with two cores. The host operating system was CentOS 5.3 64bit. The QEMU version was 0.10.2 (without kqemu). The Xen version was 3.3.1 and Domain 0 was CentOS 5.3 32bit with PAE. Testbed 2 was a Dell Optiplex GX 260 with one 2.0GHz Intel Pentium 4 CPU and 512MB memory. Xen 3.1 and Linux 2.6.18 was installed on the physical machine and the Domain 0 is CentOS 5.4.

6.1 Verifying the Static Property

An important assumption is that the control data and respective code are statically mapped into the physical memory. We used a monitoring module designed to detect legitimate control data and code modifications throughout the experiments. This enabled us to test our approach against data changes and self-modifying code in the Xen hypervisor and Domain 0. We also tested the static properties of Linux 2.6 and Windows XP 32bit kernels. In all these tests, the hypervisor and the operating systems are booted into a minimal state. The symbols used in the experiments are shown in Table 1. During the tests, we found out that during boot the control data and the code changes. For example, the physical memory of IDT is all 0 when the system first boots up. But after several seconds, it becomes non-zero and static. The reason is that the IDT table is initialized later in the boot process.

Table 1. Symbols for Xen hypervisor, Domain 0, Linux and Windows

System	Symbol	Use
Xen	idt_table	Hypervisor's Interrupt Descriptor Table
	hypercall_table	Hypervisor's Hypercall Table
	exception_table	Hypervisor's Exception Table
	_stext	Beginning of hypervisor code
	_etext	End of hypervisor code
Dom0	sys_call_table	Domain 0's System Call Table
	_text	Beginning of Domain 0's kernel code
	_etext	End of Domain 0's kernel code
Linux	idt_table	Kernel's Interrupt Descriptor Table
	sys_call_table	kernel's System Call Table
	_text	Beginning of kernel code
	_etext	End of kernel code
Windows	PCR→idt	Kernel's Interrupt Descriptor Table
	KiServiceTable	Kernel's System Call Table

6.2 Detection

To verify whether HyperCheck can detect attacks against the hypervisor, we implemented DMA attacks [40] on Xen hypervisor and then tested HyperCheck-I's response on testbed 1. We ported the HDD DMA attacks to modify the Xen hypervisor and

Domain 0. There are four attacks to Xen hypervisor and two attacks to Domain 0. We also modified the pcnet network card in QEMU to perform the DMA attack from the hardware directly. The modified pcnet NIC is used to attack Linux and Windows operating systems. There are three attacks to Linux 2.6.18 kernel and two attacks to Windows XP SP2 kernel, each targeting one control table or the code. They can modify the IDT table and other tables of the kernel. HyperCheck-I correctly detected all these attacks by reporting the contents of memory in the target machine are changed.

6.3 Monitoring Overhead

The primary source of overhead is coming from the transmission of the memory contents to the external monitoring machine. In addition, to ensure the memory contents have not been tampered with, HyperCheck needs to remain in SMM and wait until the NIC finished. Otherwise, the attacker may control the OS and modify the memory contents or the transmit descriptor in the main memory while transmitting. Initially, we measured the time to transmit a single packet varying its payload size. The packet flushed out when the Transmit Descriptor Head register (TDH) is equal to Transmit Descriptor Tail register (TDT). We calculated the elapsed time using the `rdtsc` instruction to read the time stamp before and after each operation. As expected, the time linearly increases as the size of the packet increases.

Next, we measured the bandwidth by using different packet sizes to send out a fixed amount of data: 2881 KB memory (the size of Xen code plus Domain 0 code). The result is depicted in the Figure 3: when the packet size is less than 7 KB, the time required to send the data similar to a constant value. When the packet size becomes 8KB, the overhead increases dramatically and it remains high. The reason is that the internal NIC transfer FIFO is 16KB. Therefore, when the packet size becomes 8KB or larger, the NIC cannot hold two packets in the FIFO at the same time and this introduces delay.

Since HyperCheck can be used to monitor different sized hypervisors and OSes, we measured the time required to send different amount of data and the results are in Figure 4. In this set of experiments, we use 7KB as the packet size since it introduced shortest delay in our testbed. We can see that the time also nearly linearly increased with

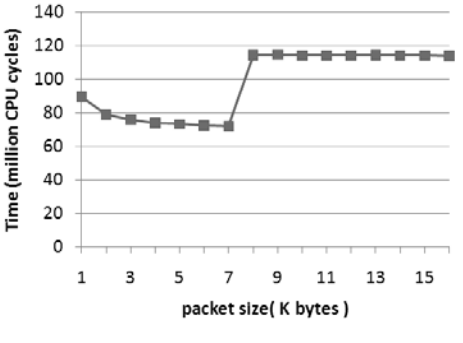


Fig. 3. Network overhead for variable packet size

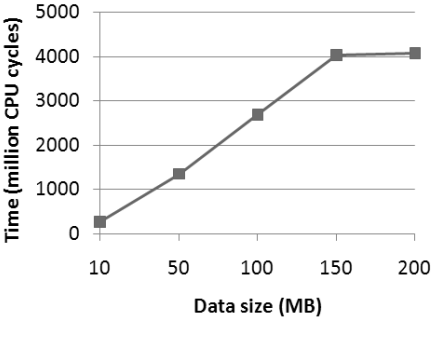


Fig. 4. Network overhead for variable data size

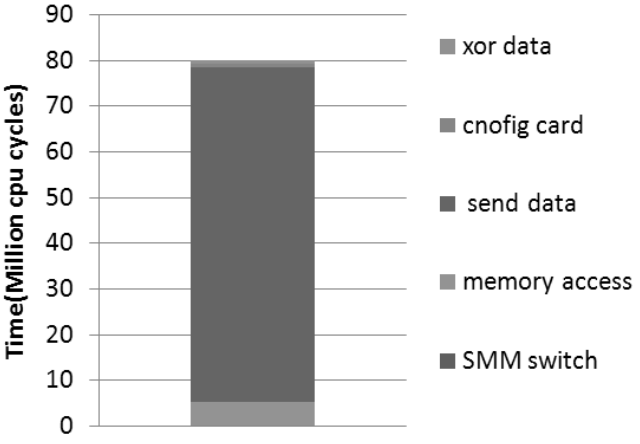


Fig. 5. Overhead of the operations in SMM

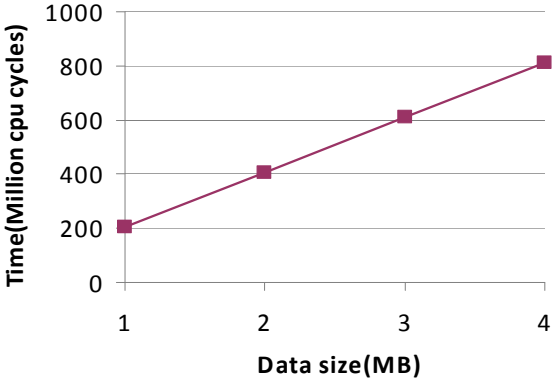


Fig. 6. Overhead of the XOR data in SMM

the amount of memory. In addition to PCI scanning, HyperCheck also triggers SMI interrupt every one second and checks the registers in SMM. To measure the overall overhead of entering SMM, executing SMM code and return from SMM, we wrote a kernel module running in Domain 0.

The tests were conducted on testbed 2 (HyperCheck-II) and each test is repeated many times. Here we present the average of the results. The overall time is composed of four parts. First, the time taken to XOR the data with the secret key. Second, the time to access the memory. Third, the time to configure the card and switch from protected mode to SMM and back. Finally, the time to send out the data through the NIC. To find out how much time was spent in each part, we wrote two more test programs. One is a dummy SMM code which does nothing but just returns from SMM to CPU protected mode. The other one does not access the main memory but just use the registers to simulate the verification of IDTR and CR3. Then we tested the running time for these two

Table 2. Time overhead of HyperCheck and other methods

		Execution Time(ms)		
Code base	Size(MB)	HC	SMM	TPM
Linux	2.0	31	203	1022
Xen+Dom0	2.7	40	274	>1022
Window XP	1.8	28	183	> 972
Hyper-V	2.4	36	244	>1022
VMWare ESXi	2.2	33	223	>1022

SMM codes. From the first one, we can get the time for switching between protected mode and SMM and then switch back. From the second one, we can get the time for the CPU computation part of the verification of IDTR and CR3.

The results are presented in Figure 5. The most of the time is spent in sending the data, which is 73 Million cycles. Next is the time to accessing the main memory : 5.28 Million cycles. Others took a very small portion. The total time is 80 Million cycles. Since the CPU of the testbed 2 is 2 GHz. Therefore, the SMM code consumes 4.0% of the CPU cycles, and takes 40 ms.

We also measured the code size of our SMM code, which is just about 300 Bytes. On the monitor machine, the overhead for reading the memory contents and comparing them with previous state took 230 ms, including 49 ms for only comparing the data. Note it is possible to reduce the time for reading the memory contents from the file, if we use pipe or other memory sharing based communication between tcpdump and the perl script.

In contrast, previous research suggests using SMM to read the memory and hashing it on the target machine. We call this SMM only method. To compare our approach with SMM only method, we wrote a program to XOR the memory in SMM with different sizes. The result is shown in Figure 6.

The time for XOR data is linearly increased with the amount of data and typically uses hundreds of Million CPU cycles. Also, we compare our approach with a TPM based approach [23] which can also be used to monitor the integrity of the kernels. The result is shown in the Table 2. HC stands for HyperCheck. We can see that the overhead of HyperCheck is one magnitude lower than SMM-only and TPM based method. For SMM-only, it has to hash the entire data to check its integrity, while HyperCheck only hashes

Table 3. Comparison between HyperCheck and other methods

	Memory	Registers	Overhead
HyperCheck	x	x	Low
SMM	x	x	High
PCI	x		Low
TPM	x	x	High

a random portion of the data and then sends the entire data out using an Ethernet card. For TPM based method, the most expensive operation is TPM quote, which alone took 972 ms. Note that the test machine of TPM based method is better than our testbed 2. An overall comparison between HyperCheck and other methods is shown in Table 3. We can see that only HyperCheck can monitor both memory and registers with low overhead.

7 Security Analysis and Limitations

HyperCheck aims to detect the modifications to the control data and the codes of the hypervisors or OS kernels. These kinds of attacks are realistic and have a significant impact on the system. HyperCheck can detect these attacks by using an Ethernet card to read the physical memory via DMA and then analyze it. For example, if the attackers control the hypervisor and make some modifications, HyperCheck can detect that change by reading the physical memory directly and compare it with previous pristine value.

In addition, HyperCheck also uses SMM to monitor CPU registers, which provides further protection. Some previous research works only rely on the symbol table in the symbol file to find the physical address of the kernel code and data. Nonetheless, there is no binding between the addresses in the symbol table and the actual physical address of these symbols [22]. For example, one potential attack is to modify the IDTR register of CPU to point to another address. Then the malware can modify the new IDT table, keeping the old one untouched. Another potential attack is to keep the IDTR register untouched, but modify the page tables of the kernel so that the virtual address in the IDTR will actually point to a different physical address. HyperCheck can detect these cases by checking CPU registers in SMM. In SMM, HyperCheck read the content of IDTR and CR3 registers used by the operating system. IDTR should never change after booting. If it changed, SMM will send a warning through the Ethernet card to the monitor machine. From CR3, HyperCheck can find the actual physical address given the virtual ones. The offset between the virtual addresses and the physical addresses should be static. If some offsets changed, HyperCheck will generate a warning too. Moreover, PCI devices including the Ethernet card alone can be cheated to get a different view of the physical memory [32]. With SMM, we could avoid this problem by checking the corresponding settings in SMM.

The network card driver of HyperCheck is put into the SMM code to avoid malicious modifications. Also, to prevent replay attacks, we use a key to hash a portion of the data randomly and then send them out to the analysis module. Since the key is private and locked in the SMRAM, the attacker cannot get it and cannot generate the same hash. Attacker can still try to disable the Ethernet card or the SMM code, but we can detect it through an out-of-band monitor, such as Dell remote access controller.

In addition, the attacker may try to launch a fake reboot attack to get a private key from the monitor machine. It can mimic the SMM NIC driver and send a request for a new key. For this event, we have two options: first, we could use Trusted Platform Module (TPM) based remote attestation to verify the running state of the target machine [23]. We only need to verify whether the OS has been started or not. If it is already started, the monitor machine should refuse to send the key. If the target machine does not have a TPM, the second method is to send another reliable reboot signal to the target machine when it asks for the key to make sure the SMM code is running.

However, HyperCheck also has its limitations. It cannot detect the changes which happen between the two consecutive memory and register scans. Although the time window between the scans is just one second in the current prototype, malware can still potentially make some changes in the time interval and restore it before the next scan. To address this problem, we could randomize the scan interval to increase the chances for detection. In addition, we could use high bandwidth devices, such as PCI Express, which is able to reach 5GT/s transfer rate [29], to minimize the scan interval.

In addition, if the memory mappings of the hypervisor do not hold the three properties (linear mapping, persistence and static nature), the current version of HyperCheck cannot deal with it. We will try to address these problems in the future.

8 Conclusions

In this paper, we introduced HyperCheck, a hardware-assisted tamper detection framework. Hypercheck is designed to protect the code integrity of software running on commodity hardware. This includes VMMs and Operating Systems. To achieve that, we rely on the CPU System Managed Mode (SMM) to securely generate and transmit the full state of the protected machine to an external server. HyperCheck does not rely on any software running on the target machine beyond BIOS. Moreover, HyperCheck is robust against attacks that aim to disable or block its operation.

To demonstrate the feasibility of our approach, we implemented two prototypes one using QEMU and another one using an Ethernet card on a commodity x86 machine. Our experimental results indicate that we can successfully identify alterations of the control data and the code on many existing systems. More specifically, we tested our approach in part of the Xen hypervisor, the Domain 0 in Xen, and the control structures of other operating systems, such as Linux and Windows. HyperCheck operation is relatively lightweight: it can produce and communicate a scan of the state of the protected software in less than 40ms.

Acknowledgements

We would like to thank the CSIS students Nelson Nazzicari, Zhaohui Wang, Quan Jia, and MeiXing Le, for their comments on our early draft. Moreover, Spyros Panagiotopoulos helped us with the DMA attack code. We also thank the anonymous RAID reviewers for their constructive comments. This material was supported in part by DARPA contract FA8650-09-C-7956, AFOSR grant FA9550-07-1-0527, and NSF grant CNS-TC 0915291.

References

[1] Adamyse, K.: Handling interrupt descriptor table for fun and profit. Phrack 59 (2002)

[2] Baliga, A., Ganapathy, V., Iftode, L.: Automatic inference and enforcement of kernel data structure invariants. In: ACSAC 2008: Proceedings of the 2008 Annual Computer Security Applications Conference, Washington, DC, USA, pp. 77–86. IEEE Computer Society, Los Alamitos (2008)

[3] Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)
[4] Bovet, D., Cesati, M.: Understanding the Linux kernel, 3rd edn. O'Reilly Media, Sebastopol (2005)
[5] BSDaemon, coideloko, and D0nAnd0n. System Management Mode Hack: Using SMM for "Other Purpose". Phrack Magazine (2008)
[6] Bulygin, Y., Samyde, D.: Chipset based approach to detect virtualization malware a.k.a. DeepWatch. Blackhat USA (2008)
[7] Burdach, M.: Digital forensics of the physical memory. Warsaw University (2005)
[8] Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Digital Investigation 1(1), 50–60 (2004)
[9] Chisnall, D.: The definitive guide to the Xen hypervisor. Prentice Hall Press, Upper Saddle River (2007)
[10] G. Coker. Xen security modules (xsm). Xen Summit (2006)
[11] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., Neugebauer, R.: Xen and the art of virtualization. In: Proceedings of the ACM Symposium on Operating Systems Principles (2003)
[12] Duflot, L., Etiemble, D., Grumelard, O.: Using CPU System Management Mode to Circumvent Operating System Security Functions. In: Proceedings of the 7th CanSecWest Conference, Citeseer (2001)
[13] Duflot, L., Etiemble, D., Grumelard, O.: Security issues related to pentium system management mode. In: Cansecwest Security Conference Core 2006 (2006)
[14] Duflot, L., Levillain, O., Morin, B., Grumelard, O.: Getting into the SMRAM: SMM Reloaded. In: CanSecWest, Vancouver, Canada (2009)
[15] Embleton, S., Sparks, S., Zou, C.: SMM rootkits: a new breed of OS independent malware. In: Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks, p. 11. ACM, New York (2008)
[16] Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proc. Network and Distributed Systems Security Symposium, pp. 191–206 (2003)
[17] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. ACPI, `http://www.acpi.info/`
[18] Hiremane, R.: Intel® Virtualization Technology for Directed I/O (Intel® VT-d). Technology© Intel Magazine 4(10) (2007)
[19] Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, Reading (2005)
[20] Intel. Intel® 64 and ia-32 architectures software developer's manual volume 1
[21] Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, p. 138. ACM, New York (2007)
[22] Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: SS 2008: Proceedings of the 17th Conference on Security Symposium, Berkeley, CA, USA, pp. 243–258. USENIX Association (2008)
[23] McCune, J., Parno, B., Perrig, A., Reiter, M., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, pp. 315–328. ACM, New York (2008)
[24] McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2010)
[25] MITRE. Cve-2007-4993

[26] Murray, D., Milos, G., Hand, S.: Improving Xen security through disaggregation. In: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 151–160. ACM, New York (2008)
[27] National Institute of Standards, NIST. National vulnerability database, http://nvd.nist.gov
[28] Payne, B., de Carbone, M., Lee, W.: Secure and flexible monitoring of virtual machines. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, pp. 385–397 (December 2007)
[29] PCI-SIG. PCI Express 2.0 Frequently Asked Questions
[30] Petroni Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - a coprocessor-based kernel runtime integrity monitor. In: SSYM 2004: Proceedings of the 13th Conference on USENIX Security Symposium, Berkeley, CA, USA, p. 13. USENIX Association (2004)
[31] Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
[32] Rutkowska, J.: Beyond the CPU: Defeating hardware based RAM acquisition. In: Proceedings of BlackHat DC 2007 (2007)
[33] Rutkowska, J., Wojtczuk, R.: Preventing and detecting Xen hypervisor subversions. Blackhat Briefings USA (2008)
[34] Sailer, R., Valdez, E., Jaeger, T., Perez, R., Van Doorn, L., Griffin, J., Berger, S.: sHype: Secure hypervisor approach to trusted virtualized systems. IBM Research Report RC23511 (2005)
[35] Schreiber, S.: Undocumented Windows 2000 secrets: a programmer's cookbook. Addison-Wesley, Reading (2001)
[36] Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, p. 350. ACM, New York (2007)
[37] Vidas, T.: The acquisition and analysis of random access memory. Journal of Digital Forensic Practice 1(4), 315–323 (2006)
[38] Wang, Z., Jiang, X., Cui, W., Ning, P.: Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 545–554. ACM, New York (2009)
[39] Wecherowski, F., collapse, c.: A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers. Phrack Magazine (2009)
[40] Wojtczuk, R.: Subverting the Xen hypervisor (2008)
[41] Wojtczuk, R., Rutkowska, J.: Attacking SMM Memory via Intel® CPU Cache Poisoning (2009)

Kernel Malware Analysis with Un-tampered and Temporal Views of Dynamic Kernel Memory

Junghwan Rhee[1], Ryan Riley[2], Dongyan Xu[1], and Xuxian Jiang[3]

[1] Purdue University
{rhee,dxu}@cs.purdue.edu
[2] Qatar University
ryan.riley@qu.edu.qa
[3] North Carolina State University
jiang@cs.ncsu.edu

Abstract. Dynamic kernel memory has been a popular target of recent kernel malware due to the difficulty of determining the status of volatile dynamic kernel objects. Some existing approaches use kernel memory mapping to identify dynamic kernel objects and check kernel integrity. The snapshot-based memory maps generated by these approaches are based on the kernel memory which may have been manipulated by kernel malware. In addition, because the snapshot only reflects the memory status at a single time instance, its usage is limited in temporal kernel execution analysis. We introduce a new runtime kernel memory mapping scheme called *allocation-driven mapping*, which systematically identifies dynamic kernel objects, including their types and lifetimes. The scheme works by capturing kernel object allocation and deallocation events. Our system provides a number of unique benefits to kernel malware analysis: (1) an un-tampered view wherein the mapping of kernel data is unaffected by the manipulation of kernel memory and (2) a temporal view of kernel objects to be used in temporal analysis of kernel execution. We demonstrate the effectiveness of allocation-driven mapping in two usage scenarios. First, we build a hidden kernel object detector that uses an un-tampered view to detect the data hiding attacks of 10 kernel rootkits that directly manipulate kernel objects (DKOM). Second, we develop a temporal malware behavior monitor that tracks and visualizes malware behavior triggered by the manipulation of dynamic kernel objects. Allocation-driven mapping enables a reliable analysis of such behavior by guiding the inspection only to the events relevant to the attack.

Keywords: Kernel memory mapping, kernel malware analysis, virtualization.

1 Introduction

Dynamic kernel memory is where the majority of kernel data resides. Operating system (OS) kernels frequently allocate and deallocate numerous dynamic objects of various types. Due to the complexity of identifying such objects at runtime, dynamic kernel memory is a source of many kernel security and reliability problems. For instance, an

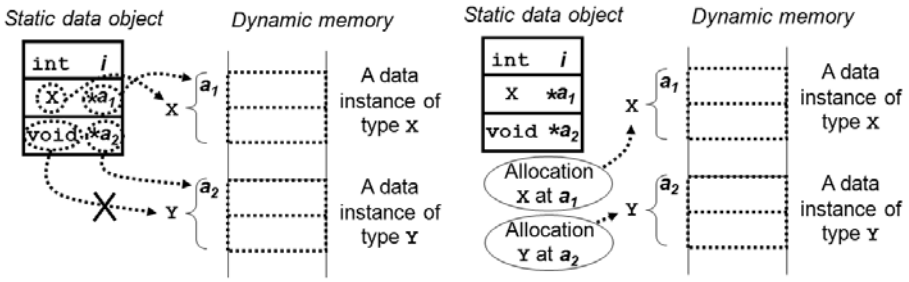
S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 178–197, 2010.

increasing amount of kernel malware targets dynamic kernel objects [4,10,18,23]; and many kernel bugs are caused by dynamic memory errors [13,27,28].

Advanced kernel malware uses stealthy techniques such as directly manipulating kernel data (i.e., DKOM [4]) or overwriting function pointers (i.e., KOH [10]) located in dynamic kernel memory. This allows attacks such as process hiding and kernel-level control flow hijacking. These anomalous kernel behaviors are difficult to analyze because they involve manipulating kernel objects that are dynamically allocated and deallocated at runtime; unlike persistent kernel code or static kernel data that are easier to locate, monitor, and protect.

To detect these attacks, some existing approaches use kernel memory mapping based on the contents of runtime memory snapshots [1,5,16] or memory access traces [23,31]. These approaches commonly identify a kernel object by projecting the type and address of a pointer onto the memory. However, such a technique may not always be accurate – for example, when an object is type cast to a generic type or when an embedded list structure is used as part of larger data types. In benign kernel execution, such inaccuracy can be corrected [5]; but it becomes a problem in malware analysis as the memory contents may have been manipulated by kernel malware. For example, a DKOM attack to hide a process may modify the `next_task` and `prev_task` pointers in the process list. This causes the process to disappear from the OS view as well as from the kernel memory map. To detect this attack, some existing approaches rely on data invariants such as that the list used for process scheduling should match the process list. However, not every data structure has an invariant. Additionally, the kernel memory map generated from a snapshot [1,5,16] reflects kernel memory status at a specific time instance. Therefore, the map is of limited usage in analyzing the kernel execution. Some mapping approaches are based on logging malware memory accesses [23,31] and thus provide temporal information. However they only cover objects accessed by the malware code and cannot properly handle certain attack patterns due to assumptions in its mapping algorithm [21].

In this paper, we present a new kernel memory mapping scheme called *allocation-driven mapping* that complements the existing approaches. Our scheme identifies dynamic kernel objects by capturing their allocations and does not rely on the runtime content of kernel memory to construct the kernel object map. As such, the map is resistant to attacks that manipulate the kernel memory. On top of our scheme, we build a hidden kernel object detector that uses the un-tampered view of kernel memory to detect DKOM data hiding attacks without requiring kernel object-specific invariants. In addition, our scheme keeps track of each kernel object's life time. This temporal property is useful in the analysis of kernel/kernel malware execution. We also build a temporal malware behavior monitor that systematically analyzes the impact of kernel malware attacks via dynamic kernel memory using a kernel execution trace. We address a challenge in the use of kernel memory mapping for temporal analysis of kernel execution: A dynamic memory address may correspond to different kernel objects at different times because of the runtime allocation and deallocation events. This problem can be handled by allocation-driven mapping. The lifetime of a dynamic kernel object naturally narrows the scope of a kernel malware analysis.



(a) Type-projection mapping (b) Allocation-driven mapping

Fig. 1. Illustration of kernel memory mapping approaches. a_1 and a_2 represent kernel memory addresses. X and Y are data types for kernel objects.

The contributions of this paper are summarized as follows:

- We present a new kernel memory mapping scheme called allocation-driven mapping that has the following properties desirable for kernel malware analysis: untampered identification of kernel objects and temporal status of kernel objects.
- We implement allocation-driven mapping at the virtual machine monitor (VMM) level. The identification and tracking of kernel objects take place in the VMM without modification to the guest OS.
- We develop a hidden kernel object detector that can detect DKOM data hiding attacks without requiring data invariants. The detector works by comparing the status of the un-tampered kernel map with that of kernel memory.
- We develop a malware behavior monitor that uses a temporal view of kernel objects in the analysis of kernel execution traces. The lifetimes of dynamic kernel objects in the view guide the analysis to the events triggered by the objects manipulated by the malware.

We have implemented a prototype of allocation-driven mapping called LiveDM (Live Dynamic kernel memory Map). It supports three off-the-shelf Linux distributions. LiveDM is designed for use in non-production scenarios such as honeypot monitoring, kernel malware profiling, and kernel debugging.

2 Background – Kernel Memory Mapping

There have been several approaches [1,5,16,23,31] that leverage kernel memory mapping to test the integrity of OS kernels and thereby detect kernel malware. These approaches (similar to garbage collection [3,19]) commonly identify kernel objects by recursively traversing pointers in the kernel memory starting from static objects. A kernel object is identified by projecting the address and type of a traversed pointer onto memory; thus, we call this mechanism *type-projection mapping*. For example, in Fig. 1(a) the mapping process starts by evaluating the pointer fields of the static data object. When the second field of this object is traversed, the type X of the pointer is projected onto the memory located in the obtained address a_1, identifying an instance of type X.

The underlying hypothesis of this mapping is that the traversed pointer's type accurately reflects the type of the projected object. In practice there are several cases when this may not be true. First, if an object allocated using a specific type is later cast to a generic type, then this mapping scheme cannot properly identify the object using that pointer. For instance, in Fig. 1(a) the third field of the static object cannot be used to identify the `Y` instance due to its generic `void*` type. Second, in modern OSes many kernel objects are linked using embedded list structures which connect the objects using list types. When these pointers are traversed, the connected objects are inaccurately identified as list objects. KOP [5] addresses these problems by generating an extended type graph using static analysis. Some other approaches rely on manual annotations.

When type-projection mapping is used against kernel malware, these problems may pose concerns as such inaccuracy can be deliberately introduced by kernel malware. In type-projection mapping, *the kernel memory map is based on the content of the kernel memory*, which may have been manipulated by the kernel malware. This property may affect the detection of kernel rootkits that hide kernel objects by directly manipulating pointers. To detect such attacks, a detector needs to rely on not only the kernel memory map but also additional knowledge that reveals the anomalous status of the hidden objects. For this purpose, several approaches [1,5,18] use data structure invariants. For example, KOP [5] detects a process hidden by the FU Rootkit [4] by using the invariant that there are two linked lists regarding process information which are supposed to match, and one of them is not manipulated by the attack. However, a data invariant is specific to semantic usage of a data structure and may not be applicable to other data structures. For type-projection mapping, it is challenging to detect data hiding attacks that manipulate a simple list structure (such as the kernel module list in Linux) without an accompanying invariant.

In general, we can categorize these approaches into two categories based on whether they make use of a static snapshot or dynamic runtime memory access trace.

2.1 Static Type-Projection Mapping

This approach uses a memory snapshot to generate a kernel memory map. SBCFI [16] constructs a map to systematically detect the violation of persistent control flow integrity. Gibraltar [1] extracts data invariants from kernel memory maps to detect kernel rootkits. A significant advantage of this approach is the low cost to generate a memory snapshot. A memory snapshot can be generated using an external monitor such as a PCI interface [1], a memory dump utility [5], or a VMM [16], and the map is generated from the snapshot later.

The memory snapshot is generated at a specific time instance (asynchronously); therefore, its usage is limited for analyzing kernel execution traces where dynamic kernel memory status varies over time. The same memory address, for example, could store different dynamic kernel objects over a period of time (through a series of deallocations and reallocations). The map cannot be used to properly determine what data was stored at that address at a specific time. We call this a *dynamic data identity problem*, and it occurs when an asynchronous kernel memory map is used for inspection of dynamic memory status along the kernel execution traces.

2.2 Dynamic Type-Projection Mapping

This mapping approach also uses the type-projection mechanism to identify kernel objects, but its input is the trace of memory accesses recorded over runtime execution instead of a snapshot. By tracking the memory accesses of malware code, this approach can identify the list of kernel objects manipulated by the malware. PoKeR [23] and Rkprofiler [31] use this approach to profile dynamic attack behavior of kernel rootkits in Linux and Windows respectively.

Since a runtime trace is used for input, this approach can overcome the asynchronous nature of static type-projection mapping. Unfortunately, current work only focuses on the data structures accessed by malware code, and may not capture other events. For example, many malware programs call kernel functions during the attack or exploit various kernel bugs, and these behaviors may appear to be part of legitimate kernel execution. In these cases, this dynamic type-projection techniques need to track all memory accesses to accurately identify the kernel objects accessed by legitimate kernel execution. Since this process is costly (though certainly possible), it is not straightforward for this approach to expand the coverage of the mapped data to all kernel objects.

3 Design of LiveDM

In this section, we first introduce the allocation-driven mapping scheme, based on which our LiveDM system is implemented. We then present key enabling techniques to implement LiveDM.

3.1 Allocation-Driven Mapping Scheme

Allocation-driven mapping is a kernel memory mapping scheme that generates a kernel object map by *capturing the kernel object allocation and deallocation events* of the monitored OS kernel. LiveDM uses a VMM in order to track the execution of the running kernel. Whenever a kernel object is allocated or deallocated, LiveDM will intercede and capture its address range and the information to derive the data type of the object subject to the event (details in Section 3.2) in order to update the kernel object map. We first present the benefits of allocation-driven mapping over existing approaches. After that we will present the techniques used to implement this mapping scheme.

First, this approach does not rely on any content of the kernel memory which can potentially be manipulated by kernel malware. Therefore, the kernel object map provides *an un-tampered view* of kernel memory wherein the identification of kernel data is not affected by the manipulation of memory contents by kernel malware. This tamper-resistant property is especially effective to detect sophisticated kernel attacks that directly manipulate kernel memory to hide kernel objects. For instance, in the type-projection mapping example (Fig. 1(a)) if the second pointer field of the static object is nullified, the `X` object cannot be identified because this object cannot be reached by recursively scanning all pointers in the memory. In practice, there can be multiple pointer references to a dynamic object. However, malware can completely isolate an

object to be hidden by tampering with all pointers pointing to the object. The address of the hidden object can be safely stored in a non-pointer storage (e.g., `int` or `char`) to avoid being discovered by the type-projection mapping algorithm while it can be used to recover the object when necessary. Many malicious programs carefully control their activities to avoid detection and prolong their stealthy operations, and it is a viable option to suspend a data object in this way temporarily and activate it again when needed [30].

In the allocation-driven mapping approach, however, this attack will not be effective. As shown in Fig. 1(b), each dynamic object is recognized upon its allocation. Therefore the identification of dynamic objects is reliably obtained and protected against the manipulation of memory contents. The key observation is that allocation-driven mapping captures the *liveness status* of the allocated dynamic kernel objects. For malware writers, this property makes it significantly more difficult to manipulate this view. In Section 6.1, we show how this mapping can be used to automatically detect DKOM data hiding attacks without using any data invariant specific to a kernel data structure.

Second, LiveDM reflects a *temporal* status of dynamic kernel objects since it captures their allocation and deallocation events. This property enables the use of the kernel object map in temporal malware analysis where temporal information, such as kernel control flow and dynamically changing data status, can be inspected to understand complicated kernel malware behavior. In Section 2.1, we pointed out that a *dynamic data identity problem* can occur when a snapshot-based kernel memory map is used for dynamic analysis. Allocation-driven mapping provides a solution to this problem by accurately tracking all allocation and deallocation events. This means that even if an object is deallocated and its memory reused for a different object, LiveDM will be able to properly track it.

Third, allocation-driven mapping does not suffer from the casting problem that occurs when an object is cast to a generic pointer because it does not evaluate pointers to construct the kernel object map. For instance, in Fig. 1(b) the `void` pointer in the third field of the static data object does not hinder the identification of the `Y` instance because this object is determined by capturing its allocation. However, we note that another kind of casting can pose a problem: If an object is allocated using a generic type and it is cast to a specific type later, allocation-driven mapping will detect the earlier generic type. However, our study in Section 5 shows that this behavior is unusual in Linux kernels.

There are a number of challenges in implementing the LiveDM system based on allocation-driven mapping. For example, kernel memory allocation functions do not provide a simple way to determine the type of the object being allocated.[1] One solution is to use static analysis to rewrite the kernel code to deliver the allocation types to the VMM, but this would require the construction of a new type-enabled kernel, which is not readily applicable to off-the-shelf systems. Instead, we use a technique that derives data types by using runtime context (i.e., call stack information). Specifically, this technique systematically captures code positions for memory allocation calls by using virtual machine techniques (Section 3.2) and translates them into data types so that OS kernels can be transparently supported without any change in the source code.

[1] Kernel level memory allocation functions are similar to user level ones. The function `kmalloc`, for example, does not take a type but a size to allocate memory.

3.2 Techniques

We employ a number of techniques to implement allocation-driven mapping. At the conceptual level, LiveDM works as follows. First, a set of kernel functions (such as `kmalloc`) are designated as kernel memory allocation functions. If one of these functions is called, we say that an allocation event has occurred. Next, whenever this event occurs at runtime, the VMM intercedes and captures the allocated memory address range and the code location calling the memory allocation function. This code location is referred to as a *call site* and we use it as a unique identifier for the allocated object's type at runtime. Finally, the source code around each call site is analyzed offline to determine the type of the kernel object being allocated.

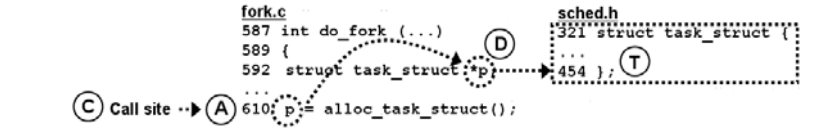
Runtime kernel object map generation. At runtime, LiveDM captures all allocation and deallocation events by interceding whenever one of the allocation/deallocation functions is called. There are three things that need to be determined at runtime: (1) the call site, (2) the address of the objected allocated or deallocated, and (3) the size of the allocated object.

To determine the call site, LiveDM uses the return address of the call to the allocation function. In the instruction stream, the return address is the address of the instruction after the call instruction. The captured call site is stored in the kernel object map so that the type can be determined during offline source code analysis.

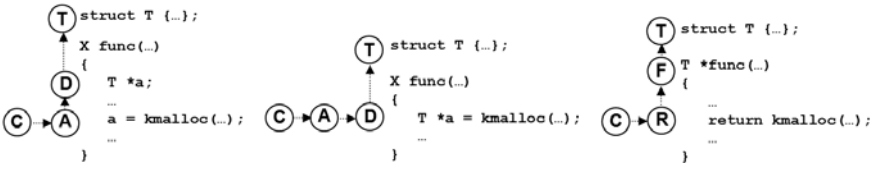
The address and size of objects being allocated or deallocated can be derived from the arguments and return value. For an allocation function, the size is typically given as a function argument and the memory address as the return value. For a deallocation function, the address is typically given as a function argument. These values can be determined by the VMM by leveraging *function call conventions*.[2] Function arguments are delivered through the stack or registers, and LiveDM captures them by inspecting these locations at the entry of memory allocation/deallocation calls. To capture the return value, we need to determine where the return value is stored and when it is stored there. Integers up to 32-bits as well as 32-bit pointers are delivered via the `EAX` register and all values that we would like to capture are either of those types. The return value is available in this register when the allocation function returns to the caller. In order to capture the return values at the correct time the VMM uses a virtual stack. When a memory allocation function is called, the return address is extracted and pushed on to this stack. When the address of the code to be executed matches the return address on the stack, the VMM intercedes and captures the return value from the `EAX` register.

Offline automatic data type determination. The object type information related to kernel memory allocation events is determined using static analysis of the kernel source code offline. Fig. 2(a) illustrates a high level view of our method. First, the allocation call site (C) of a dynamic object is mapped to the source code `fork.c:610` using debugging information found in the kernel binary. This code assigns the address of the allocated memory to a pointer variable at the left-hand side (LHS) of the assignment statement (A). Since this variable's type can represent the type of the allocated memory,

[2] A function call convention is a scheme to pass function arguments and a return value. We use the conventions for the x86 architecture and the `gcc` compiler [8].



(a) A high level view of static code analysis



(b) Case 1 (c) Case 2 (d) Case 3

Fig. 2. Static code analysis. C: a call site, A: an assignment, D: a variable declaration, T: a type definition, R: a return, and F: a function declaration

it is derived by traversing the declaration of this pointer (D) and the definition of its type (T). Specifically, during the compilation of kernel source code, a parser sets the dependencies among the internal representations (IRs) of such code elements. Therefore, the type can be found by following the dependencies of the generated IRs.

For type resolution, we enumerate several patterns in the allocation code as shown in Fig. 2(b), 2(c), and 2(d). Case 1 is the typical pattern (C→A→D→T) as just explained. In Case 2, the definition (D) and allocation (A) occur in the same line. The handling of this case is very similar to that of Case 1. Case 3, however, is unlike the first two cases. The pattern in Case 3 does not use a variable to handle the allocated memory address, rather it directly returns the value generated from the allocation call. When a call site (C) is converted to a return statement (R), we determine the type of the allocated memory using the type of the returning function (F). In Fig. 2(d), this pattern is presented as C→R→F→T.

Prior to static code analysis, we generate the set of information about these code elements to be traversed (i.e., C, A, D, R, F, and T) by compiling the kernel source code with the compiler that we instrumented (Section 4).

4 Implementation

Allocation-driven mapping is general enough to work with an OS that follows the standard function call conventions (e.g., Linux, Windows, etc.). Our prototype, LiveDM, supports three off-the-shelf Linux OSes of different kernel versions: Fedora Core 6 (Linux 2.6.18), Debian Sarge (Linux 2.6.8), and Redhat 8 (Linux 2.4.18).

LiveDM can be implemented on any software virtualization system, such as VMware (Workstation and Player) [29], VirtualBox [26], and Parallels [14]. We choose the QEMU [2] with KQEMU optimizer for implementation convenience.

In the kernel source code, many wrappers are used for kernel memory management, some of which are defined as macros or inline functions and others as regular functions.

Macros and inline functions are resolved as the core memory function calls at compile time by a preprocessor; thus, their call sites are captured in the same way as core functions. However, in the case of regular wrapper functions, the call sites will belong to the wrapper code.

To solve this problem, we take two approaches. If a wrapper is used only a few times, we consider that the type from the wrapper can indirectly imply the type used in the wrapper's caller due to its limited use. If a wrapper is widely used in many places (e.g., `kmem_cache_alloc` – a slab allocator), we treat it as a memory allocation function. Commodity OSes, which have mature code quality, have a well defined set of memory wrapper functions that the kernel and driver code commonly use. In our experience, capturing such wrappers, in addition to the core memory functions, can cover the majority of the memory allocation and deallocation operations.

We categorize the captured functions into four classes: (1) page allocation/free functions, (2) `kmalloc/kfree` functions, (3) `kmem_cache_alloc/free` functions (slab allocators), and (4) `vmalloc/vfree` functions (contiguous memory allocators). These sets include the well defined wrapper functions as well as the core memory functions. In our prototype, we capture about 20 functions in each guest kernel. The memory functions of an OS kernel can be determined from its design specification (e.g., the Linux Kernel API) or kernel source code.

Automatic translation of a call site to a data type requires a kernel binary that is compiled with a debugging flag (e.g., `-g` to `gcc`) and whose symbols are not stripped. Modern OSes, such as Ubuntu, Fedora, and Windows, generate kernel binaries of this form. Upon distribution, typically the stripped kernel binaries are shipped; however, unstripped binaries (or symbol information in Windows) are optionally provided for kernel debugging purposes. The experimented kernels of Debian Sarge and Redhat 8 are not compiled with this debugging flag. Therefore, we compiled the distributed source code and generated the debug-enabled kernels. These kernels share the same source code with the distributed kernels, but the offset of the compiled binary code can be slightly different due to the additional debugging information.

For static analysis we use a `gcc` [8] compiler (version 3.2.3) that we instrumented to generate IRs for the source code of the experimented kernels. We place hooks in the parser to extract the abstract syntax trees for the code elements necessary in the static code analysis.

5 Evaluation

In this section, we evaluate the basic functionality of LiveDM with respect to the identification of kernel objects, casting code patterns, and the performance of allocation-driven mapping. The guest systems are configured with 256MB RAM and the host machine has a 3.2Ghz Pentium D CPU and 2GB of RAM.

Identifying dynamic kernel objects. To demonstrate the ability of LiveDM to inspect the runtime status of an OS kernel, we present a list of important kernel data structures captured during the execution of Debian Sarge OS in Table 1. These data structures manage the key OS status such as process information, memory mapping of each process, and the status of file systems and network which are often targeted by kernel malware

Table 1. A list of core dynamic kernel objects and the source code elements used to derive their data types in static analysis. (OS: Debian Sarge).

	Call Site	Declaration	Data Type	Case	#Objects
Task/Sig	kernel/fork.c:248	kernel/fork.c:243	task_struct	1	66
	kernel/fork.c:801	kernel/fork.c:795	sighand_struct	1	63
	fs/exec.c:601	fs/exec.c:587	sighand_struct	1	1
	kernel/fork.c:819	kernel/fork.c:813	signal_struct	1	66
Memory	arch/i386/mm/pgtable.c:229	arch/i386/mm/pgtable.c:229	pgd_t	2	54
	kernel/fork.c:433	kernel/fork.c:431	mm_struct	1	47
	kernel/fork.c:559	kernel/fork.c:526	mm_struct	1	7
	kernel/fork.c:314	kernel/fork.c:271	vm_area_struct	1	149
	mm/mmap.c:923	mm/mmap.c:748	vm_area_struct	1	1004
	mm/mmap.c:1526	mm/mmap.c:1521	vm_area_struct	1	5
	mm/mmap.c:1722	mm/mmap.c:1657	vm_area_struct	1	48
	fs/exec.c:402	fs/exec.c:342	vm_area_struct	1	47
File system	kernel/fork.c:677	kernel/fork.c:654	files_struct	1	54
	kernel/fork.c:597	kernel/fork.c:597	fs_struct	2	53
	fs/file_table.c:76	fs/file_table.c:69	file	1	531
	fs/buffer.c:3062	fs/buffer.c:3062	buffer_head	2	828
	fs/block_dev.c:232	fs/block_dev.c:232	bdev_inode	2	5
	fs/dcache.c:692	fs/dcache.c:689	dentry	1	4203
	fs/inode.c:112	fs/inode.c:107	inode	1	1209
	fs/namespace.c:55	fs/namespace.c:55	vfsmount	2	16
	fs/proc/inode.c:93	fs/proc/inode.c:90	proc_inode	1	237
	drivers/block/ll_rw_blk.c:1405	drivers/block/ll_rw_blk.c:1405	request_queue_t	2	18
	drivers/block/ll_rw_blk.c:2950	drivers/block/ll_rw_blk.c:2945	io_context	1	10
Network	net/socket.c:279	net/socket.c:278	socket_alloc	1	12
	net/core/sock.c:617	net/core/sock.c:613	sock	1	3
	net/core/dst.c:125	net/core.dst.c:119	dst_entry	1	5
	net/core/neighbour.c:265	net/core/neighbour.c:254	neighbour	1	1
	net/ipv4/tcp_ipv4.c:134	net/ipv4/tcp_ipv4.c:133	tcp_bind_bucket	2	4
	net/ipv4/fib_hash.c:586	net/ipv4/fib_hash.c:461	fib_node	1	9

and kernel bugs [13,15,16,17,18,23,27,28]. Kernel objects are recognized using allocation call sites shown in column **Call Site** during runtime. Using static analysis, this information is translated into the data types shown in column **Data Type** by traversing the allocation code and the declaration of a pointer variable or a function shown in column **Declaration**. Column **Case** shows the kind of the allocation code pattern described in Section 3.2. The number of the identified objects for each type in the inspected runtime status is presented in column **#Objects**. At that time instance, LiveDM identified total of 29488 dynamic kernel objects with their data types derived from 231 allocation code positions.

In order to evaluate the accuracy of the identified kernel objects, we build a reference kernel where we modify kernel memory functions to generate a log of dynamic kernel objects and run this kernel in LiveDM. We observe that the dynamic objects from the log accurately match the live dynamic kernel objects captured by LiveDM. To check the type derivation accuracy, we manually translate the captured call sites to data types by traversing kernel source code as done by related approaches [5,7]. The derived types at the allocation code match the results from our automatic static code analysis.

Code patterns casting objects from generic types to specific types. In Section 3.1, we discussed that allocation-driven mapping has no problem handling the situation where a specific type is cast to a generic type, but casting from generic types to specific types can

be a problem. In order to estimate how often this type of casting occurs, we manually checked all allocation code positions where the types of kernel objects are derived for the inspected status. We checked for the code pattern that memory is allocated using a generic pointer and then the address is cast to the pointer of a more specific type. Note that this pattern does not include the use of generic pointers for generic purposes. For example, the use of void or integer pointers for bit fields or buffers is a valid use of generic pointers. Another valid use is kernel memory functions that internally handle pre-typed memory using generic pointers to retail it to various types. We found 25 objects from 10 allocation code positions (e.g., `tty_register_driver` and `vc_allocate`) exhibiting this behavior at runtime. Such objects are not part of the core data structures shown in Table 1, and they account for only 0.085% of all objects. Hence we consider them as non-significant corner cases. Since the code positions where this casting occurs are available to LiveDM, we believe that the identification of this behavior and the derivation of a specific type can be automated by performing static analysis on the code after the allocation code.

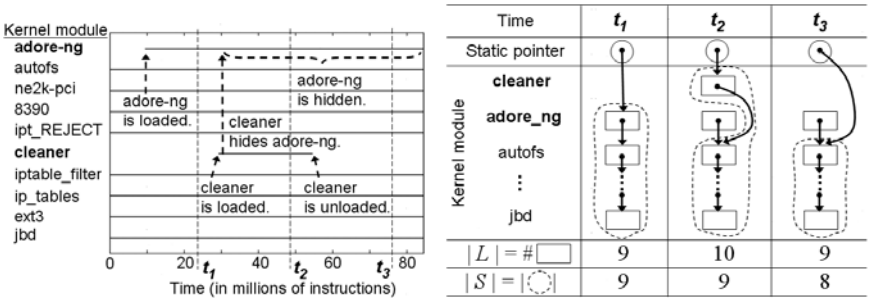
Performance of allocation-driven mapping. Since LiveDM is mainly targeted for non-production environments such as honeypots and kernel debugging systems, performance is not a primary concern. Still, we would like to provide a general idea of the cost of allocation-driven mapping. In order to measure the overhead to generate a kernel object map at runtime, we ran three benchmarks: compiling the kernel source code, UnixBench (Byte Magazine Unix Benchmark 5.1.2), and nbench (BYTEmark* Native Mode Benchmark version 2). Compared to unmodified QEMU, our prototype incurs (in the worst case) 41.77% overhead for Redhat 8 (Linux 2.4) and 125.47% overhead for Debian Sarge (Linux 2.6). For CPU intensive workload such as nbench, the overhead is near zero because the VMM rarely intervenes. However, applications that use kernel services requiring dynamic kernel memory have higher overhead. As a specific example, compiling the Linux kernel exhibited an overhead of 29% for Redhat 8 and 115.69% for Debian Sarge. It is important to note that these numbers measure overhead when compared to an unmodified VMM. Software based virtualization will add additional overhead as well. For the purpose of inspecting fine-grained kernel behavior in non-production environments, we consider this overhead acceptable. The effects of overhead can even be minimized in a production environment by using decoupled analysis [6].

6 Case Studies

We present two kernel malware analysis systems built on top of LiveDM: a hidden kernel object detector and a temporal malware behavior monitor. These systems highlight the new properties of allocation-driven mapping which are effective for detection and analysis of kernel malware attacks.

6.1 Hidden Kernel Object Detector

One problem with static type-projection approaches is that they are not able to detect dynamic kernel object manipulation without some sort of data invariant. In this section



(a) Temporal live status of kernel modules based on allocation-driven mapping.

(b) Live set (L) and scanned set (S) for kernel modules at t_1, t_2, and t_3.

Fig. 3. Illustration of the kernel module hiding attack by `cleaner` rootkit. Note that the choice of t_1, t_2, and t_3 is for the convenience of showing data status and irrelevant to the detection. This attack is detected based on the difference between L and S.

we present a hidden kernel object detector built on top of LiveDM that does not suffer from this limitation.

Leveraging the un-tampered view. Some advanced DKOM-based kernel rootkits hide kernel objects by simply removing all references to them from the kernel's dynamic memory. We model the behavior of this type of DKOM data hiding attack as a data anomaly in a list. If a dynamic kernel object does not appear in a kernel object list, then it is orphaned and hence an anomaly. As described in Section 3.1, allocation-driven mapping provides an un-tampered view of the kernel objects not affected by manipulation of the actual kernel memory content. Therefore, if a kernel object appears in the LiveDM-generated kernel object map but cannot be found by traversing the kernel memory, then that object has been hidden. More formally, for a set of dynamic kernel objects of a given data type, a live set L is the set of objects found in the kernel object map. A scanned set S is the set of kernel objects found by traversing the kernel memory as in the related approaches [1,5,16]. If L and S do not match, then a data anomaly will be reported.

This process is illustrated in the example of `cleaner` rootkit that hides the `adore-ng` rootkit module (Fig. 3). Fig. 3(a) presents the timeline of this attack using the lifetime of kernel modules. Fig. 3(b) illustrates the detailed status of kernel modules and corresponding L and S at three key moments. Kernel modules are organized as a linked list starting from a static pointer variable. When the `cleaner` module is loaded after the `adore-ng` module, it modifies the linked list to bypass the `adore-ng` module entry (shown at t_2). Therefore, when the `cleaner` module is unloaded, the `adore-ng` module disappears from the module list (t_3). At this point in time the scanned set S based on static type-projection mapping has lost the hidden module, but the live set L keeps the view of all kernel modules alive. Therefore, the monitor can detect a hidden kernel module due to the condition, $|L| \neq |S|$.

Detecting DKOM data hiding attacks. There are two dynamic kernel data lists which are favored by rootkits as attack targets: the kernel module list and the process control

Table 2. DKOM data hiding rootkit attacks that are automatically detected by comparing LiveDM-generated view (L) and kernel memory view (S)

Rootkit Name	$\|L\|$ - $\|S\|$	Manipulated Data		Operating System	Attack Vector
		Type	Field		
hide_lkm	# of hidden modules	module	next	Redhat 8	/dev/kmem
fuuld	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	/dev/kmem
cleaner	# of hidden modules	module	next	Redhat 8	LKM
modhide	# of hidden modules	module	next	Redhat 8	LKM
hp 1.0.0	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	LKM
linuxfu	# of hidden PCBs	task_struct	next_task, prev_task	Redhat 8	LKM
modhide1	1 (rootkit self-hiding)	module	next	Redhat 8	LKM
kis 0.9 (server)	1 (rootkit self-hiding)	module	next	Redhat 8	LKM
adore-ng-2.6	1 (rootkit self-hiding)	module	list.next, list.prev	Debian Sarge	LKM
ENYELKM 1.1	1 (rootkit self-hiding)	module	list.next, list.prev	Debian Sarge	LKM

block (PCB) list.[3] However other linked list-based data structures can be similarly supported as well. The basic procedure is to generate the live set L and periodically generate and compare with the scanned set S. We tested 8 real-world rootkits and 2 of our own rootkits (linuxfu and fuuld) previously used in [12,21,23], and these rootkits commonly hide kernel objects by directly manipulating the pointers of such objects. LiveDM successfully detected all these attacks just based on the data anomaly from kernel memory maps and the results are shown in Table 2.

In the experiments, we focus on a specific attack mechanism – data hiding via DKOM – rather than the attack vectors – how to overwrite kernel memory – or other attack features of rootkits for the following reason. There are various attack vectors including the ones that existing approaches cannot handle and they can be easily utilized. Specifically, we acknowledge that the rootkits based on loadable kernel module (LKM) can be detected by code integrity approaches [22,24] with the white listing scheme of kernel modules. However, there exist alternate attack vectors such as /dev/mem, /dev/kmem devices, return-oriented techniques [11,25], and unproven code in third-party kernel drivers which can elude existing kernel rootkit detection and prevention approaches. We present the DKOM data hiding cases of LKM-based rootkits as part of our results because these rootkits can be easily converted to make use of these alternate attack vectors.

We also include results for two other rootkits that make use of these advanced attack techniques. hide_lkm and fuuld in Table 2 respectively hide kernel modules and processes without any kernel code integrity violation (via /dev/kmem) purely based on DKOM, and current rootkit defense approaches cannot properly detect these attacks. However, our monitor effectively detects all DKOM data hiding attacks regardless of attack vectors by leveraging LiveDM-generated kernel object map. Allocation-driven mapping can uncover the hidden object even in more adversary scenarios. For example, if a simple linked list having no data invariant is directly manipulated without violating kernel code integrity, LiveDM will still be able to detect such an attack and uncover the specific hidden object.

In the experiments that detect rootkit attacks, we generate and compare L and S sets every 10 seconds. When a data anomaly occurs, the check is repeated in 1 second. (The

[3] A process control block (PCB) is a kernel data structure containing administrative information for a particular process. Its data type in Linux is task_struct.

repeated check ensures that a kernel data structure was not simply in an inconsistent state during the first scan.) If the anomaly persists, then we consider it as a true positive. With this monitoring policy, we successfully detected all tested DKOM hiding attacks without any false positives or false negatives.

We note that while this section focuses on data hiding attacks based on DKOM, data hiding attacks without manipulating data (such as rootkit code that filters system call results) may also be detected using the LiveDM system. Instead of comparing the un-tampered LiveDM-generated view with the scanned view of kernel memory, one could simply compare the un-tampered view with the user-level view of the system.

6.2 Temporal Malware Behavior Monitor

Kernel rootkit analysis approaches based on dynamic type-projection are able to perform temporal analysis of a running rootkit. One problem with these approaches, however, is that they are only able to track malware actions that occur from injected rootkit code. If a rootkit modifies memory indirectly through other means such as legitimate kernel functions or kernel bugs, these approaches are unable to follow the attack. Allocation-driven mapping does not share this weakness. To further illustrate the strength of allocation-driven mapping, we built a temporal malware behavior monitor (called a temporal monitor or a monitor below for brevity) that uses a kernel object map in temporal analysis of a kernel execution trace.

In this section, we highlight two features that allocation-driven mapping newly provides. First, allocation-driven mapping enables the *use of a kernel object map covering all kernel objects in temporal analysis*; therefore for any given dynamic kernel object we can inspect how it is being used in the dynamic kernel execution trace regardless of the accessing code (either legitimate or malicious), which is difficult for both static and dynamic type-projection approaches. Second, the data lifetime in allocation-driven mapping lets the monitor *avoid the dynamic data identity problem* (Section 2.1) which can be faced by an asynchronous memory map.

Systematic visualization of malware influence via dynamic kernel memory. Our monitor systematically inspects and visualizes the influence of kernel malware attacks targeting dynamic kernel memory. To analyze this dynamic attack behavior, we generate a full system trace including the kernel object map status, the executed code, and the memory accesses during the experiments of kernel rootkits. When a kernel rootkit attack is launched, if it violates kernel code integrity, the rootkit code is identified by using our previous work, NICKLE [22]. Then the temporal monitor systematically identifies all targets of rootkit memory writes by searching the kernel object map. If the attack does not violate code integrity, the proposed technique in the previous section or any other approach can be used to detect the dynamic object under attack. The identified objects then become the *causes* of malware behavior and their *effects* are systematically visualized by searching the original and the modified kernel control flow triggered by such objects. For each object targeted by the rootkit, there are typically multiple behaviors using its value. Among those, this monitor samples a pair of behaviors caused by the same code,

Table 3. The list of kernel objects manipulated by `adore-ng` rootkit. (OS: Redhat 8).

Runtime Identification		Offline Data Type Interpretation	
Call Site	**Offset**	**Type / Object (Static, Module object)**	**Field**
`fork.c:610`	`0x4,12c,130`	**`task_struct`** (Case (1))	**`flags,uid,euid`**
`fork.c:610`	`0x134,138,13c`	**`task_struct`** (Case (1))	**`suid,fsuid,gid`**
`fork.c:610`	`0x140,144,148`	**`task_struct`** (Case (1))	**`egid,sgid,fsgid`**
`fork.c:610`	`0x1d0`	**`task_struct`** (Case (1))	**`cap_effective`**
`fork.c:610`	`0x1d4`	**`task_struct`** (Case (1))	**`cap_inheritable`**
`fork.c:610`	`0x1d8`	**`task_struct`** (Case (1))	**`cap_permitted`**
`generic.c:436`	`0x20`	**`proc_dir_entry`** (Case (2))	**`get_info`**
(Static object)		`proc_root_inode_operations`	`lookup`
(Static object)		`proc_root_operations`	`readdir`
(Static object)		`unix_dgram_ops`	`recvmsg`
(Module object)		`ext3_dir_operations`	`readdir`
(Module object)		`ext3_file_operations`	`write`

the latest one before the attack and the earliest one after the attack, and presents them for a comparison.

As a running example in this section, we will present the analysis of the attacks by the `adore-ng` rootkit. This rootkit is chosen because of its advanced malware behavior triggered by dynamic objects; and other rootkits can be analyzed in a similar way. Table 3 lists the kernel objects that the `adore-ng` rootkit tampers with. In particular, we focus on two specific attack cases using dynamic objects: (1) The first case is the manipulation of a PCB (T_3) for privilege escalation and (2) the second case is the manipulation of a function pointer in a dynamic `proc_dir_entry` object (P_1) to hijack kernel control flow. Fig. 4 presents a detailed view of kernel control flow and the usage of the targeted dynamic kernel memory in the attacks. The X axis shows the execution time, and kernel control flow is shown at top part of this figure. The space below shows the temporal usage of dynamic memory at the addresses of T_3 and P_1 before and after rootkit attacks. Thick horizontal lines represent the lifetime of kernel objects which are temporally allocated at such addresses. $+$ and $\times$ symbols below such lines show the read and write accesses on corresponding objects. The aforementioned analysis process is illustrated as solid arrows. From the times when T_3 and P_1 are manipulated (shown as dotted circles), the monitor scans the execution trace backward and forward to find the code execution that consumes the values read from such objects (i.e., $+$ symbols).

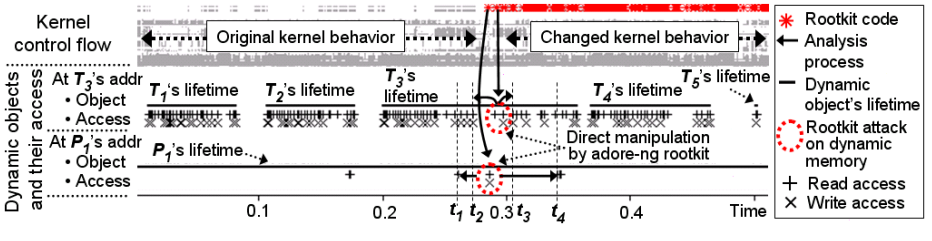
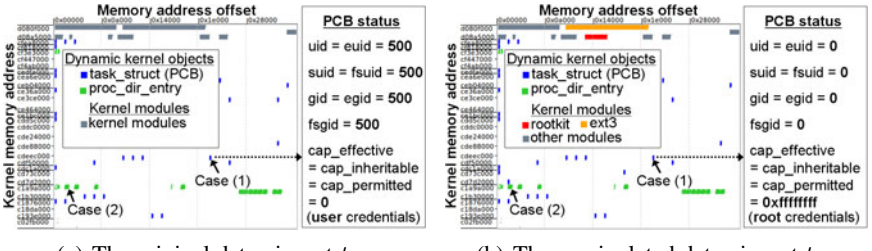


Fig. 4. Kernel control flow (top) and the usage of dynamic memory (below) at the addresses of T_3 (Case (1)) and P_1 (Case (2)) manipulated by the `adore-ng` rootkit. Time is in billions of kernel instructions.

Selecting semantically relevant kernel behavior using data lifetime. Our monitor inspects dynamic memory states in the temporal execution trace and as such we face the *dynamic data identity problem* described in Section 3.1. The core of the problem is that one memory address may correspond with multiple objects over a period of time. This problem can be solved if the lifetime of the inspected object is available because the monitor can filter out irrelevant kernel behaviors triggered by other kernel objects that share the same memory address. For example, in Fig. 4, we observe the memory for T_3 is used for four other PCBs (i.e., T_1, T_2, T_4, and T_5) as well in the history of kernel execution. Simply relying on the memory address to analyze the trace can lead to finding kernel behavior for *all five PCBs*. However, the monitor limits the inspected time range to the lifetime of T_3 and select only semantically relevant behaviors to T_3. Consequently it can provide a reliable inspection of runtime behavior only relevant to attacks.

Other kernel memory mapping approaches commonly cannot handle this problem properly. In static type-projection, when two kernel objects from different snapshots are given we cannot determine whether they represent the same data instance or not even though their status is identical because such objects may or may not be different data instances depending on whether memory allocation/deallocation events occur between the generation of such snapshots. Dynamic type-projection mapping is only based on malware instructions, and thus does not have information about allocation and deallocation events which occur during legitimate kernel execution.

Case (1): Privilege escalation using direct memory manipulation. In order to demonstrate the effectiveness of our temporal monitor we will discuss two specific attacks employed by `adore-ng`. The first is a privilege escalation attack that works by modifying the user and group ID fields of the PCB. The PCB is represented by T_3 in Fig. 4. To present the changed kernel behavior due to the manipulation of T_3, the temporal monitor finds the latest use of T_3 before the attack (at t_2) and the earliest use of it after the attack (at t_3). The data views at such times are presented in Fig. 5(a) and 5(b) as 2-dimensional memory maps where a kernel memory address is represented as the combination of the address in Y axis and the offset in X axis. These views present kernel objects relevant to this attack before and after the attack. The manipulated PCB is marked with "Case (1)" in the views and the values of its fields are shown in the box on the right side of each view (PCB status). These values reveal a stealthy rootkit behavior that changes the identity of

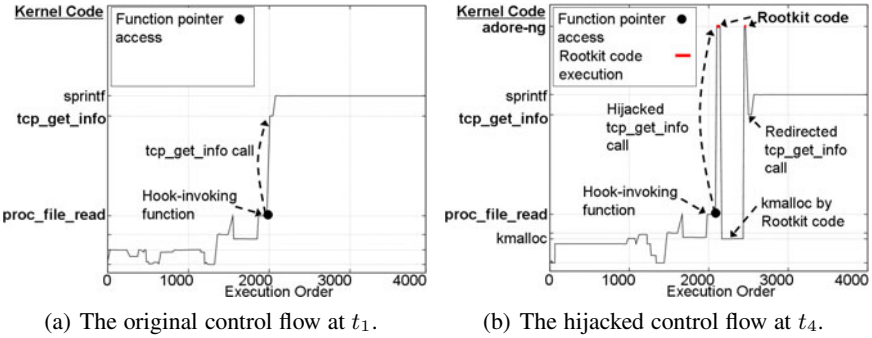


(a) The original data view at t_2. (b) The manipulated data view at t_3.

Fig. 5. Kernel data view before and after the `adore-ng` rootkit attack

a user process by directly patching its PCB (DKOM). Before the attack (Fig. 5(a)), the PCB has the credentials of an ordinary user whose user ID is 500. However, after the attack, Fig. 5(b) shows the credentials of the root user. This direct transition of its status between two accounts is abnormal in conventional operating system environments. `su` or `sudo` allow privileged operations by forking a process to retain the original identity. Hence we determine that this is a case of privilege escalation that illegally permits the root privilege to an ordinary user.

Case (2): Dynamic kernel object hooking. The next `adore-ng` attack hijacks kernel code execution by modifying a function pointer and this attack is referred to as Kernel Object Hooking (KOH) [10]. This behavior is observed when the influence of a manipulated function pointer in P_1 (see Fig. 4) is inspected. To select only the behaviors caused by this object, the monitor guides the analysis to the lifetime of P_1. The temporal monitor detects several behaviors caused by reading this object and two samples are chosen among those to illustrate the change of kernel behavior by comparison: the latest original behavior before the attack (at t_1) and the earliest changed behavior after the attack (at t_4). The monitor generates two kernel control flow graphs at these samples, each for a period of 4000 instructions. Fig. 6(a) and 6(b) present how this manipulated function pointer affects runtime kernel behavior. The Y axis presents kernel code; thus, the fluctuating graphs show various code executed at the corresponding time of X axis. A hook-invoking function (`proc_file_read`) reads the function pointer and calls the hook code pointed to by it. Before the rootkit attack, the control flow jumps to a legitimate kernel function `tcp_get_info` which calls `sprintf` after that as shown in Fig. 6(a). However, after the hook is hijacked, the control flow is redirected to the rootkit code which calls `kmalloc` to allocate its own memory, then comes back to the original function (Fig. 6(b)).



(a) The original control flow at t_1. (b) The hijacked control flow at t_4.

Fig. 6. Kernel control flow view before and after the `adore-ng` rootkit attack

7 Discussion

Since LiveDM operates in the VMM beneath the hardware interface, we assume that kernel malware cannot directly access LiveDM code or data. However, it can exhibit potentially obfuscating behavior to confuse the view seen by LiveDM. Here we describe

several scenarios in which malware can affect LiveDM and our counter-strategies to detect them.

First, malware can implement its own custom memory allocators to bypass LiveDM observation. This attack behavior can be detected based on the observation that any memory allocator must use internal kernel data structures to manage memory regions or its memory may be accidentally re-allocated by the legitimate memory allocator. Therefore, we can detect unverified memory allocations by comparing the resource usage described in the kernel data structures with the amount of memory being tracked by LiveDM. Any deviance may indicate the presence of a custom memory allocator.

In a different attack strategy, malware could manipulate valid kernel control flow and jump into the body of a memory allocator without entering the function from the beginning. This behavior can be detected by extending LiveDM to verify that the function was entered properly. For example, the VMM can set a flag when a memory allocation function is entered and verify the flag before the function returns by interceding before the return instruction(s) of the function. If the flag was not set prior to the check, the VMM detects a suspicious memory allocation.

8 Related Work

Static type-projection mapping has been widely used in the defense against kernel malware attacks. SBCFI [16] detects persistent manipulations to the kernel control flow graph by using kernel memory maps. Gibraltar [1] derives data invariants based on a kernel memory map to detect kernel malware. KOP [5] improves the accuracy of mapping using extended type graph based on static analysis in addition to memory analysis. Complementing these approaches, allocation-driven mapping provides an un-tampered view of kernel objects where their identification is not affected by kernel malware's manipulation of the kernel memory content. It also accurately reflects the temporal status of dynamic kernel memory, which makes it applicable to temporal analysis of kernel/kernel malware execution.

PoKeR [23] and Rkprofiler [31] use dynamic type-projection mapping generated from rootkit instructions to understand the rootkit behavior. Since only rootkit activity is used as the input to generate a kernel memory map, this approach can only cover the kernel objects directly manipulated by rootkit code. Moreover, there exist the attacks that are difficult to be analyzed by these profilers because rootkits can use various resource such as hardware registers to find the attack targets [21].

KernelGuard (KG) [20] is a system that prevents DKOM-based kernel rootkits by monitoring and shepherding kernel memory accesses. It identifies kernel objects to be monitored by scanning the kernel memory using data structure-specific policies enforced at the VMM level. Similar to type-projection mapping, KG's view of kernel memory is based on the runtime kernel memory content which is subject to malware manipulation. As such, KG's reliability can be improved by adopting LiveDM as the underlying kernel memory mapping mechanism.

LiveDM involves techniques to capture the location, type, and lifetime of individual dynamic kernel objects, which can be described as belonging to the area of virtual machine introspection [9].

9 Conclusion

We have presented allocation-driven mapping, a kernel memory mapping scheme, and LiveDM, its implementation. By capturing the kernel objects' allocation and deallocation events, our scheme provides an un-tampered view of kernel objects that will not be affected by kernel malware's manipulation of kernel memory content. The LiveDM-generated kernel object map accurately reflects the status of dynamic kernel memory and tracks the lifetimes of all dynamic kernel objects. This temporal property is highly desirable in temporal kernel execution analysis where both kernel control flow and dynamic memory status can be analyzed in an integrated fashion. We demonstrate the effectiveness of the LiveDM system by developing a hidden kernel object detector and a temporal malware behavior monitor and applying them to a corpus of kernel rootkits.

Acknowledgements. We thank the anonymous reviewers for their insightful comments. This research was supported, in part, by the Air Force Research Laboratory (AFRL) under contract FA8750-09-1-0224 and by the National Science Foundation (NSF) under grants 0716444, 0852131, 0855036 and 0855141. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the AFRL or NSF.

References

1. Baliga, A., Ganapathy, V., Iftode, L.: Automatic Inference and Enforcement of Kernel Data Structure Invariants. In: Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008), pp. 77–86 (2008)
2. Bellard, F.: QEMU: A Fast and Portable Dynamic Translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46 (2005)
3. Boehm, H.J., Weiser, M.: Garbage Collection in an Uncooperative Environment. Software, Practice and Experience (1988)
4. Butler, J.: DKOM (Direct Kernel Object Manipulation), `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`
5. Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., Jiang, X.: Mapping Kernel Objects to Enable Systematic Integrity Checking. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS 2009 (2009)
6. Chow, J., Garfinkel, T., Chen, P.M.: Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In: Proceedings of 2008 USENIX Annual Technical Conference, USENIX 2008 (2008)
7. Cozzie, A., Stratton, F., Xue, H., King, S.T.: Digging For Data Structures. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (2008)
8. Free Software Foundation: The GNU Compiler Collection, `http://gcc.gnu.org/`
9. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the 10th Annual Network and Distributed Systems Security Symposium, NDSS 2003 (2003)
10. Hoglund, G.: Kernel Object Hooking Rootkits (KOH Rootkits), `http://www.rootkit.com/newsread.php?newsid=501`
11. Hund, R., Holz, T., Freiling, F.C.: Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In: Proceedings for the 18th USENIX Security Symposium (2009)

12. Lin, Z., Riley, R.D., Xu, D.: Polymorphing Software by Randomizing Data Structure Layout. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 107–126. Springer, Heidelberg (2009)
13. MITRE Corp.: Common Vulnerabilities and Exposures, http://cve.mitre.org/
14. Parallels: Parallels, http://www.parallels.com/
15. Petroni, N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot - A Coprocessor-based Kernel Runtime Integrity Monitor. In: Proceedings for the 13th USENIX Security Symposium (August 2004)
16. Petroni, N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007 (2007)
17. Petroni, N.L., Walters, A., Fraser, T., Arbaugh, W.A.: FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory. Digital Investigation Journal 3(4), 197–210 (2006)
18. Petroni, Jr. N.L., Fraser, T., Walters, A., Arbaugh, W.A.: An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In: Proceedings of the 15th Conference on USENIX Security Symposium, USENIX-SS 2006 (2006)
19. Polishchuk, M., Liblit, B., Schulze, C.W.: Dynamic Heap Type Inference for Program Understanding and Debugging. In: Proceedings of the 34th Annual Symposium on Principles of Programming Languages. ACM, New York (2007)
20. Rhee, J., Riley, R., Xu, D., Jiang, X.: Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In: International Conference on Availability, Reliability and Security, ARES 2009 (2009)
21. Rhee, J., Xu, D.: LiveDM: Temporal Mapping of Dynamic Kernel Memory for Dynamic Kernel Malware Analysis and Debugging. Tech. Rep. 2010-02, CERIAS (2010)
22. Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 1–20. Springer, Heidelberg (2008)
23. Riley, R., Jiang, X., Xu, D.: Multi-Aspect Profiling of Kernel Rootkit Behavior. In: Proceedings of the 4th European Conference on Computer Systems (Eurosys 2009) (April 2009)
24. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: Proceedings of 21st Symposium on Operating Systems Principles (SOSP 2007). ACM, New York (2007)
25. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007), pp. 552–561. ACM, New York (2007)
26. Sun Microsystems, Inc: VirtualBox, http://www.virtualbox.org/
27. The Month of Kernel Bugs archive, http://projects.info-pull.com/mokb/
28. US-CERT: Vulnerability Notes Database, http://www.kb.cert.org/vuls/
29. VMware, Inc.: VMware Virtual Machine Technology, http://www.vmware.com/
30. Wei, J., Payne, B.D., Giffin, J., Pu, C.: Soft-Timer Driven Transient Kernel Control Flow Attacks and Defense. In: Proceedings of the 24th Annual Computer Security Applications Conference, ACSAC 2008 (December 2008)
31. Xuan, C., Copeland, J.A., Beyah, R.A.: Toward Revealing Kernel Malware Behavior in Virtual Execution Environments. In: Proceedings of 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009), pp. 304–325 (2009)

Bait Your Hook: A Novel Detection Technique for Keyloggers*

Stefano Ortolani[1], Cristiano Giuffrida[1], and Bruno Crispo[2]

[1] Vrije Universiteit, De Boelelaan 1081, 1081HV Amsterdam, The Netherlands
{ortolani,giuffrida}@cs.vu.nl
[2] University of Trento, Via Sommarive 14, 38050 Povo, Trento, Italy
crispo@disi.unitn.it

Abstract. Software keyloggers are a fast growing class of malware often used to harvest confidential information. One of the main reasons for this rapid growth is the possibility for unprivileged programs running in user space to eavesdrop and record all the keystrokes of the users of the system. Such an ability to run in unprivileged mode facilitates their implementation and distribution, but, at the same time, allows to understand and model their behavior in detail. Leveraging this property, we propose a new detection technique that simulates carefully crafted keystroke sequences (the bait) in input and observes the behavior of the keylogger in output to univocally identify it among all the running processes. We have prototyped and evaluated this technique with some of the most common free keyloggers. Experimental results are encouraging and confirm the viability of our approach in practical scenarios.

Keywords: Keylogger, Malware, Detection, Black-box.

1 Introduction

Keyloggers are implanted on a machine to intentionally monitor the user activity by logging keystrokes and eventually sending them to a third party. While they are sometimes used for legitimate purposes (i.e. child computer monitoring), keyloggers are often maliciously exploited by attackers to steal confidential information. Many credit card numbers and passwords have been stolen using keyloggers [17,19], which makes them one of the most dangerous types of spyware. Keyloggers can be implemented as tiny hardware devices or more conveniently in software. A software acting as a keylogger can be implemented by means of two different techniques: as a kernel module or as a user-space process. It is important to notice that, while a kernel keylogger requires a privileged access to the system, a user-space keylogger can easily rely on documented sets of unprivileged API commonly available on modern operating systems. A user can

* This work has been partially funded by the EU FP7 IP Project MASTER (contract no. 216917) and by the PRIN project "Paradigmi di progettazione completamente decentralizzati per algoritmi autonomici".

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 198–217, 2010.

be easily deceived in installing it, and, since no special permission is required, the user can erroneously regard it as a harmless piece of software. On the contrary, kernel-level keyloggers require a considerable effort and knowledge for an effective and bug-free implementation. It is therefore no surprise that 95% of the existing keyloggers are user-space keyloggers [9]. Despite the number of frauds exploiting keyloggers (i.e. identity theft, password leakage, etc.) has increased rapidly in recent years, not many effective and efficient solutions have been proposed to address this problem. Preventing keyloggers to be implanted without limiting the behavior of the user is hardly an option in real-world scenarios. Traditional defensive mechanisms use fingerprinting or heuristic-based strategies similar to those used to detect viruses and worms. Unfortunately, results have been poor due to keyloggers' small footprint and their ability to hide.

In this paper, we propose a new approach to detect keyloggers running as unprivileged user-space processes. Our technique is entirely implemented in an unprivileged process. As a result, our solution is portable, unintrusive, easy to install, and yet very effective. Moreover, the proposed detection technique does not depend on the internal structure of the keylogger or the particular set of APIs used to capture the keystrokes. On the contrary, our solution is of general applicability, since it is based on behavioral characteristics common to all the keyloggers. We have prototyped our approach and evaluated it against the most common free keyloggers [15]. Our approach has proven effective in all the cases. We have also evaluated the impact of false positives and false negatives in practical scenarios.

The structure of the paper is as follows. We first present our approach and compare it to analogous solutions (Sec. 2). We then detail the architecture of our solution in Sec. 3 and evaluate the resulting prototype in Sec. 4. Sec. 5 discusses how a keylogger may counter our approach, and why our underlying model would still be valid. We conclude with related work in Sec. 6 and final remarks in Sec. 7.

2 Our Approach

Common misuse-based approaches rely on the ability to build a profile of the malicious system activity. Once the profile is available, any behavior that matches any known malicious patterns is reported as anomalous. However, applying analogous approaches to malware detection is not a trivial task. Building a malicious profile requires the ability to identify what a malicious behavior is. Unfortunately, such a behavior is normally triggered by factors that are neither easy to analyze nor feasible to control. In our approach, we explore the opposite direction. Rather than targeting the general case, we focus on designing a detection technique for a very peculiar category of malware, the keyloggers. In contrast to many other malicious programs, a keylogger has a very well defined behavior that is easy to model. In its simplest form, a keylogger eavesdrops each keystroke issued by the user and logs the content on a file on the disk. In this scenario, the events triggering the malicious activities are always known in advance and could be reproduced and controlled to some extent.

Our model is based on these observations and investigates the possibility to isolate the keylogger in a controlled environment, where its behavior is directly exposed to the detection system. Our technique involves controlling the keystroke events that the keylogger receives in input, and constantly monitoring the I/O activities generated by the keylogger in output. To detect malicious behavior, we leverage the intuition that the relationship between the input and output of the controlled environment can be modeled for most keyloggers with very good approximation. Whatever transformations the keylogger performs, a characteristic pattern observed in the keystroke events in input shall somehow be reproduced in the I/O activity in output. When the input and the output are controlled, there is potential to identify common patterns and trigger a detection. Furthermore, if we can select and enforce the input pattern preventively, we can better avoid possible evasion attempts. The key advantage of our approach is that it is centered around a black-box model that completely ignores the internals of a keylogger. As a result, our technique can deal with a large number of keyloggers transparently and has the potential to realize a fully-unprivileged detection system.

Our approach completely ignores the content of the input and the output data, and focuses exclusively on their distribution. Limiting the approach to a quantitative analysis enables the ability to implement the detection technique with only unprivileged mechanisms, as we will better illustrate later. The underlying model adopted, however, presents additional challenges. First, we must carefully deal with possible data transformations that may introduce quantitative differences between the input and the output patterns. Second, the technique should be robust with respect to quantitative similarities identified in the output patterns of other legitimate system processes. In the following, we discuss how our approach deals with these challenges.

3 Architecture

Our design is based on five different components as depicted in Fig. 1: injector, monitor, pattern translator, detector, pattern generator. The operating system at the bottom deals with the details of I/O and event handling. The *OS Domain* does not expose all the details to the upper levels without using privileged API calls. As a result, the injector and the monitor operate at another level of abstraction, the *Stream Domain*. At this level, keystroke events and the bytes output by a process appear as a stream emitted at a particular rate.

The task of the injector is to inject a keystroke stream to simulate the behavior of a user typing keystrokes on the keyboard. Similarly, the monitor records a stream of bytes to constantly capture the output behavior of a particular process. A stream representation is only concerned with the distribution of keystrokes or bytes emitted over a given window of observation, without entailing any additional qualitative information. The injector receives the input stream from the pattern translator, that acts as bridge between the *Stream Domain* and the *Pattern Domain*. Similarly, the monitor delivers the output stream recorded to the

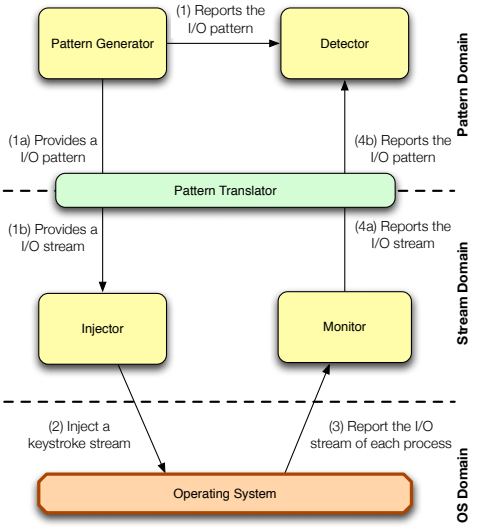


Fig. 1. The prototype's architecture divided in components and domains

pattern translator for further analysis. In the *Pattern Domain*, the input stream and the output stream are both represented in a more abstract form, termed *Abstract Keystroke Pattern* (AKP). A pattern in the AKP form is a discretized and normalized representation of a stream. Adopting a compact and uniform representation is advantageous for several reasons. First, we allow the pattern generator to exclusively focus on generating an input pattern that follows a desired distribution of values. Details on how to inject a particular distribution of keystrokes into the system are offloaded to the pattern translator and the injector. Second, the same input pattern can be reused to produce and inject several input streams with different properties but following the same underlying distribution. Finally, the ability to reason over abstract representations simplifies the role of the detector that only receives an input pattern and an output pattern and makes the final decision whether detection should be triggered.

3.1 Injector

The role of the injector is to inject the input stream into the system, mimicking the behavior of a simulated user at the keyboard. By design, the injector must satisfy several requirements. First, it should only rely on unprivileged API calls. Second, it should be capable of injecting keystrokes at variable rates to match the distribution of the input stream. Finally, the resulting series of keystroke events produced should be no different than those generated by a user at the keyboard. In other words, no user-space keylogger should be somehow able to distinguish the two types of events. To address all these issues, we leverage the same technique employed in automated testing. On Windows-based operating systems,

for example, this functionality is provided by the API call `SendInput`, available for several versions of the OS. All the other OSes supporting the X11 window server, the same functionality is available via the API call `XTestFakeKeyEvent`, part of the `XTEST` extension library.

3.2 Monitor

The monitor is responsible to record the output stream of all the running processes. As done for the injector, we allow only unprivileged API calls. In addition, we favor strategies to perform realtime monitoring with minimal overhead and the best level of resolution possible. Finally, we are interested in application-level statistics of I/O activities, to avoid dealing with filesystem-level caching or other potential nuisances. Fortunately, most modern operating systems provide unprivileged API calls to access performance counters on a per-process basis. On all the versions of Windows since Windows NT 4.0, this functionality is provided by the Windows Management Instrumentation (WMI). In particular, the performance counters of each process are made available via the class `Win32_Process`, that supports an efficient query-based interface. All the performance counters are constantly maintained up-to-date by the kernel. In WMI, the counter `WriteTransferCount` contains the total number of bytes the process wrote since its creation. To construct the output stream of a given process, the monitor queries this piece of information at regular time intervals, and records the number of bytes written since the last query every time. The proposed technique is obviously tailored to Windows-based operating systems. Nonetheless, we point out that similar strategies can be realized in other OSes. Linux, for instance, supports analogous performance counters since the 2.6.19 version.

3.3 Pattern Translator

The role of the pattern translator is to transform an AKP into a stream and vice-versa, given a set of target configuration parameters. A pattern in the AKP form can be modeled as a sequence of samples originated from a stream sampled with a uniform time interval. A sample P_i of a pattern P is an abstract representation of the number of keystrokes emitted during the time interval i. Each sample is stored in a normalized form rescaled in the interval $[0, 1]$, where 0 and 1 reflect the predefined minimum and maximum number of keystrokes in a given time interval, respectively. To transform an input pattern into a keystroke stream, the pattern translator considers the following configuration parameters:

N – the number of samples in the pattern.
T – the constant time interval between any two successive samples.
K_{min} – the minimum predefined number of keystrokes per sample allowed.
K_{max} – the maximum predefined number of keystrokes per sample allowed.

When transforming an input pattern in the AKP form into an input stream, the pattern translator generates, for each time interval i, a keystroke stream

with an average keystroke rate $\overline{R}_i = \frac{P_i \cdot (K_{max} - K_{min}) + K_{min}}{T}$. The iteration is repeated N times to cover all the samples in the original pattern. A similar strategy is adopted when transforming an output byte stream into a pattern in the AKP form. The pattern translator reuses the same parameters employed in the generation phase and similarly assigns $P_i = \frac{\overline{R}_i \cdot T - K_{min}}{K_{max} - K_{min}}$ where $\overline{R}_i$ is the average keystroke rate measured in the time interval i.

The translator assumes a correspondence between keystrokes and bytes and treats them equally as base units of the input and output stream, respectively. This assumption does not always hold in practice and the detection algorithm has to consider any possible scale transformation between the input and the output pattern. We discuss this and other potential transformations in more detail in Sec. 3.4.

3.4 Detector

The success of our detection algorithm lies in the ability to infer a cause-effect relationship between the keystroke stream injected in the system and the I/O behavior of a keylogger process, or, more specifically, between the respective patterns in AKP form. While one must examine every candidate process in the system, the detection algorithm operates on a single process at a time, identifying whether there is a strong similarity between the input pattern and the output pattern obtained from the analysis of the I/O behavior of the target process. Specifically, given a predefined input pattern and an output pattern of a particular process, the goal of the detection algorithm is to determine whether there is a match in the patterns and the target process can be identified as a keylogger with good probability.

The first step in the construction of a detection algorithm comes down to the adoption of a suitable metric to measure the similarity between two given patterns. In principle, the AKP representation allows for several possible measures of dependence that compare two discrete sequences and quantify their relationship. In practice, we rely on a single correlation measure motivated by the properties of the two patterns. The proposed detection algorithm is based on the Pearson product-moment correlation coefficient (PCC), the first formally defined correlation measure and still one of the most widely used [18]. Given two discrete sequences described by two patterns P and Q with N samples, the PCC is defined as [18]:

$$r = \frac{\mathrm{cov}(P,Q)}{\sigma_p \cdot \sigma_q} = \frac{\sum_{i=1}^{N}(P_i - \overline{P})(Q_i - \overline{Q})}{\sqrt{\sum_{i=1}^{N}(P_i - \overline{P})^2}\sqrt{\sum_{i=1}^{N}(Q_i - \overline{Q})^2}} \tag{1}$$

where $\mathrm{cov}(P,Q)$ is the sample covariance, σ_p and σ_q are sample standard deviations, and $\overline{P}$ and $\overline{Q}$ are sample means. The PCC has been widely used as an index to measure bivariate association for different distributions in several applications including pattern recognition, data analysis, and signal processing [5]. The values given by the PCC are always symmetric and ranging between -1 and 1, with 0 indicating no correlation and 1 or -1 indicating complete direct (or inverse) correlation. To measure the degree of association between two given

patterns we are here only interested in positive values of correlation. Hereafter, we will always refer to its absolute value.

Our interest in the PCC lies in its appealing mathematical properties. In contrast to many other correlation metrics, the PCC measures the strength of a linear relationship between two series of samples, ignoring any non-linear association. In the context of our detection algorithm, a linear dependence well approximates the relationship between the input pattern and an output pattern produced by a keylogger. The basic intuition is that a keylogger can only make local decisions on a per-keystroke basis with no knowledge about the global distribution. Thus, in principle, whatever the decisions, the resulting behavior will linearly approximate the original input stream injected into the system.

In detail, the PCC is resilient to any change in location and scale, namely no difference can be observed in the correlation coefficient if every sample P_i of any of the two patterns is transformed into $a \cdot P_i + b$, where a and b are arbitrary constants. This is important for a number of reasons. Ideally, the input pattern and an output pattern will be an exact copy of each other if every keystroke injected is replicated as it is in the output of a keylogger process. In practice, different data transformations performed by the keylogger can alter the original structure in several ways. First, a keylogger may encode each keystroke in a sequence of one or more bytes. Consider, for example, a keylogger encoding each keystroke using 8-bit ASCII codes. The output pattern will be generated examining a stream of raw bytes produced by the keylogger as it stores keystrokes one byte at a time. Now consider the exact same case but with keystrokes stored using a different encoding, e.g. 2 bytes per keystroke. In the latter case, the pattern will have the same shape as the former one, but its scale will be twice as much. Fortunately, as explained earlier, the transformation in scale will not affect the correlation coefficient and the PCC will report the same value in both cases. Similar arguments are valid for keyloggers using a variable-length representation to store keystrokes. This scenario occurs, for instance, when a keylogger uses special byte sequences to encode particular classes of keystrokes or encrypts keystrokes with a variable number of bytes. Even under these circumstances, the resulting data transformation can still be approximated as linear. The scale invariance property makes also the approach robust to keyloggers that drop a limited number of keystrokes while logging. For example, many keyloggers refuse to record keystrokes that do not directly translate into alphanumeric characters. In this case, under the assumption that keystrokes in the input stream are uniformly distributed by type, the resulting output pattern will only contain each generated keystroke with a certain probability p. This can be again approximated as rescaling the original pattern by p, with no significant effect on the original value of the PCC.

An interesting application of the location invariance property is the ability to mitigate the effect of buffering. When the keylogger uses a fixed-size buffer whose size is comparable to the number of keystrokes injected at each time interval, it is easy to show that the PCC is not significantly affected. Consider, for example, the case when the buffer size is smaller than the minimum number of keystrokes K_{min}. Under this assumption, the buffer is completely flushed out

at least once per time interval. The number of keystrokes left in the buffer at each time interval determines the number of keystrokes missing in the output pattern. Depending on the distribution of samples in the input pattern, this number would be centered around a particular value z. The statistical meaning of the value z is the average number of keystrokes dropped per time interval. This transformation can be again approximated by a location transformation of the original pattern by a factor of $-z$, which again does not affect the value of the PCC. The last example shows the importance of choosing an appropriate K_{min} when the effect of fixed-size buffers must also be taken into account. As evident from the examples discussed, the PCC is robust when not completely resilient to several possible data transformations. Nevertheless, there are other known fundamental factors that may affect the size of the PCC and could possibly complicate the interpretation of the results. A taxonomy of these factors is proposed and thoroughly discussed in [8]. We will briefly discuss some of these factors here to analyze how they affect our design. This is crucial to avoid common pitfalls and unexpectedly low correlation values that underestimate the true relationship between two patterns possibly generating false negatives.

A first important factor to consider is the possible lack of linearity. Although the several cases presented only involve linear or pseudo-linear transformations, non-linearity might still affect our detection system in the extreme case of a keylogger performing aggressive buffering. A representative example in this category is a keylogger flushing out to disk an indefinite-size buffer at regular time intervals. While we experimented this circumstance to rarely occur in practice, we have also adopted standard strategies to deal with this scenario effectively. In our design, we exploit the observation that the non-linear behavior is known in advance and can be modeled with good approximation.

Following the solution suggested in [8], we transform both patterns to eliminate the source of non-linearity before computing the PCC. To this end, assuming a sufficiently large number of samples N is available, we examine peaks in the output pattern and eliminate non-informative samples when we expect to see the effect of buffering in action. At the same time, we aggregate the corresponding samples in the input pattern accordingly and gain back the ability to perform a significative linear analysis using the PCC over the two normalized patterns. The advantage of this approach is that it makes the resulting value of the PCC practically resilient to buffering. The only potential shortcoming is that we may have to use larger windows of observation to collect a sufficient number of samples N for our analysis.

Another fundamental factor to consider is the number of samples collected. While we would like to shorten the duration of the detection algorithm as much as possible, there is a clear tension between the length of the patterns examined and the reliability of the resulting value of the PCC. A very small number of samples can lead to unstable or inaccurate results. A larger number of samples is beneficial especially whenever one or more other disturbing factors are to be expected. As reported in [8], selecting a larger number of samples could, for example, reduce the adverse effect of outliers or measurement errors. The

detection algorithm we have implemented in our detector, relies entirely on the PCC to estimate the correlation between an input and an output pattern. To determine whether a given PCC value should trigger a detection, a thresholding mechanism is used. We discuss how to select a suitable threshold empirically in Sec. 4. Our detection algorithm is conceived to infer a causal relationship between two patterns by analyzing their correlation. Admittedly, experience shows that correlation cannot be used to imply causation in the general case, unless valid assumptions are made on the context under investigation [2]. In other words, to avoid false positives in our detection strategy, strong evidence shall be collected to infer with good probability that a given process is a keylogger. The next section discusses in detail how to select a robust input pattern and minimize the probability of false detections.

3.5 Pattern Generator

Our pattern generator is designed to support several possible pattern generation algorithms. More specifically, the pattern generator can leverage any algorithm producing a valid input pattern in AKP form. In this section, we present a number of pattern generation algorithms and discuss their properties.

First important issue to consider is the effect of variability in the input pattern. Experience shows that correlations tend to be stronger when samples are distributed over a wider range of values [8]. In other words, the more the variability in the given distributions, the more stable and accurate the resulting PCC computed. This suggests that a robust input pattern should contain samples spanning the entire target interval $[0, 1]$. The level of variability in the resulting input stream is also similarly influenced by the range of keystroke rates used in the pattern translation process. The higher the range delimited by the minimum keystroke rate and maximum keystroke rate, the more reliable the results.

The adverse effect of low variability in the input pattern can be best understood when analyzing the mathematical properties of the PCC. The correlation coefficient reports high values of correlation when the two patterns tend to grow apart from their respective means on the same side with proportional intensity. As a consequence, the more closely to their respective means the patterns are distributed, the less stable and accurate the resulting PCC.

In the extreme case of no variability, that is when a constant distribution is considered, the standard deviation is 0 and the PCC is not even defined. This suggests that a robust pattern generation algorithm should never consider constant or low-variability patterns. Moreover, when a constant pattern is generated from the output stream, our detection algorithm assigns an arbitrary correlation score of 0. This is still coherent under the assumption that the selected input pattern presents a reasonable level of variability, and poor correlation should naturally be expected when comparing with other low-variability patterns. A robust pattern generation algorithm should allow for a minimum number of false positives and false negatives at detection time. As far as false negatives are

concerned, we have already discussed some of the factors that affect the PCC and may increase the number of false detections in Sec. 3.4.

About false positives, when the chosen input pattern happens to closely resemble the I/O behavior of some benign process in the system, the PCC may report a high value of correlation for that process and trigger a false detection. For this reason, it is important to focus on input patterns that have little chances of being confused with output patterns generated by regular system processes. Fortunately, studies show that the correlation between different realistic I/O workloads for PC users is generally considerably low different I/O workloads for PC users is generally considerably low over small time intervals [11]. The results presented in [11] are derived from 14 traces collected over a number of months in realistic environments used by different categories of users. The authors show that the value of correlation given by the PCC over 1 minute of I/O activity is only 0.0462 on average and never exceeds 0.0708 for any two given traces. These results suggest that the I/O behavior of one or more given processes is in general very poorly correlated with other different I/O distributions.

Another property of interest concerning the characteristics of common I/O workloads is self-similarity. Experience shows that the I/O traffic is typically self-similar, namely that its distribution and variability are relatively insensitive to the size of the sampling interval [11]. For our analysis, this suggests that variations in the time interval T will not heavily affect the sample distribution in the output pattern and thereby the values of the resulting PCC. This scale-invariant property is crucial to allow for changes in the parameter T with no considerable variations in the number of potential false positives generated at detection time. While most pattern generation algorithms with the properties discussed so far should produce a relatively small number of false positives in common usage scenarios, we are also interested in investigating pattern generation algorithms that attempt to minimize the number of false positives for a given target workload.

The problem of designing a pattern generation algorithm that minimizes the number of false positives under a given known workload can be modeled as follows. We assume that traces for the target workload can be collected and converted into a series of patterns (one for each process running on the system) of the same length N. All the patterns are generated to build a valid training set for the algorithm. Under the assumption that the traces collected are representative of the real workload available at detection time, our goal is to design an algorithm that learns the characteristics of the training data and generates a maximally uncorrelated input pattern. Concretely, the goal of our algorithm is to produce an input pattern of length N that minimizes the average PCC measured against all the patterns in the training set. Without any further constraints on the samples of the target input pattern, it can be shown that this problem is a non-trivial non-linear optimization problem. In practice, we can relax the original problem by leveraging some of the assumptions discussed earlier. As motivated before, a robust input pattern should present samples distributed over a wide range of values. To assume the widest range possible, we can arbitrarily constraint the

series of samples to be uniformly distributed over the target interval $[0, 1]$. This is equivalent to consider a set of N samples of the form:

$$S = \left\{0, \frac{1}{N-1}, \frac{2}{N-1}, \ldots, \frac{N-2}{N-1}, 1\right\} \quad . \tag{2}$$

When the N samples are constrained to assume all the values from the set S, the optimization problem comes down to finding the particular permutation of values that minimizes the average PCC. This problem is a variant of the standard assignment problem for N objects and N tasks, where each particular pairwise assignment yields a known cost and the ultimate goal is to minimize the sum of all the costs involved [14].

In our scenario, the objects can be modeled by the samples in the target set S and the tasks reflect the N slots in the input pattern each sample has to be assigned to. In addition, the cost of assigning a sample S_i from the set S to a particular slot j is $c(i, j) = \sum_t \frac{(S_i - \overline{S})(P_{t_j} - \overline{P}_t)}{\sigma_s \cdot \sigma_{p_t}}$, where P_t are the patterns in the training set, and $\overline{S}$ and σ_s are the constant mean and standard distribution of the samples in S, respectively. The cost value $c(i, j)$ reflects the value of a single addendum in the resulting expression of the average PCC we want to minimize. The formulation of the cost value has been simplified assuming constant number of samples N and constant number of patterns in the training set. Unfortunately, this problem cannot be easily addressed by leveraging well-known algorithms that solve the linear assignment problem in polynomial time [14]. In contrast to the standard formulation, we are not interested in the global minimum of the sum of the cost values. Such an approach would indeed attempt to find a pattern that results in an average PCC maximally close to -1. In contrast, the ultimate goal of our analysis is to produce a maximally uncorrelated pattern, thereby aiming at an average PCC as close to 0 as possible. This problem can be modeled as an assignment problem with side constraints.

Prior research has shown how to transform this particular problem into an equivalent quadratic assignment problem (QAP) that can be very efficiently solved with a standard QAP solver when the global minimum is known in advance [13]. In our solution, we have implemented a similar approach limiting the approach to a maximum number of iterations to guarantee convergence in bounded time since the minimum value of the PCC is not known in advance. In practice, for a reasonable number of samples N and a modest training set, we found that this is rarely a concern. The algorithm can usually identify the optimal pattern in a bearable amount of time. To conclude, we now more formally propose two classes of pattern generation algorithms for our generator. First, we are interested in workload-aware generation algorithms. For this class, we focus on the optimization algorithm we have just introduced, assuming a number of representative traces have been made available for the target workload.

Moreover, we are interested in workload-agnostic pattern generation algorithms. With no assumption made on the nature of the workload, they are more

generic and easier to implement. In this class, we propose the following algorithms:

Random (RND). Every sample is generated at random with no additional constraints. This is the simplest pattern generation algorithm.

Random with fixed range (RFR). The pattern is a random permutation of a series of samples uniformly distributed over the interval $[0, 1]$. This algorithm attempts to maximize the amount of variability in the input pattern.

Impulse (IMP). Every sample $2i$ is assigned the value of 0 and every sample $2i + 1$ is assigned the value of 1. This algorithm attempts to produce an input pattern with maximum variance while minimizing the duration of idle periods.

Sine Wave (SIN). The pattern generated is a discrete sine wave distribution oscillating between 0 and 1 with the first sample having the value of 1. The sine wave grows or drops with a fixed step of 0.1 at every sample. This algorithm explores the effect of constant increments (and decrements) in the input pattern.

4 Evaluation

To demonstrate the viability of our approach and evaluate the proposed detection technique, we implemented a prototype system based on the ideas described in the paper. Our prototype is entirely written in `C#` and runs as an unprivileged application for the Windows operating system.

In the following, we present several experiments to evaluate our approach. The ultimate goal is to understand the effectiveness of our technique and whether it can be used in realistic settings. We experimented our prototype with many publicly available keyloggers. We have also developed our own keylogger to evaluate the effect of special features or particular conditions more throughly. Finally, we have collected traces for different realistic PC workloads to evaluate the strength of our approach in real-life scenarios. We ran all of our experiments on PCs with a 2.53 GHz Core 2 Duo processor, 4 GB memory, and 7200 rpm SATA II hard drives. Every test was performed under Windows XP Professional SP3, while the workload traces were gathered from a number of PCs running several different versions of Windows.

4.1 Keylogger Detection

To evaluate the ability to detect real-world keyloggers, we experimented all the keyloggers from the top monitoring free software list [15], an online repository continuously updated with reviews and latest developments in the area. At the moment of writing, eight keyloggers were listed in the free software list. To carry out the experiments, we manually installed each keylogger, launched our detection system for $N \cdot T$ ms, and recorded the results.

In the experiments, we used arbitrary settings for the threshold and the parameters N, T, K_{min}, K_{max}. The reason is that we observed the same results for several reasonable combinations of parameters in most cases. We have also solely selected the RFR algorithm as the pattern generation algorithm for the experiments. More details on how to select a pattern generation algorithm and

Table 1. Detection results for the keyloggers used in the evaluation. PCC's threshold set to 0.80.

Keylogger	Detection	Notes
Refog Keylogger Free 5.4.1	✔	uses focus-based buffering
Best Free Keylogger (BFK) 1.1	✔	
Iwantsoft Free Keylogger 3.0	✔	
Actual Keylogger 2.3	✔	uses focus-based buffering
Revealer Keylogger Free 1.4	✔	uses focus-based buffering
Virtuoza Free Keylogger 2.0	✔	uses time-based buffering
Quick Keylogger 3.0.031	N/A	unable to test it properly
Tesline KidLogger 1.4	N/A	unable to test it properly

tune parameters and threshold in the general case are given in Sec. 4.2 and Sec. 4.3. Table 1 shows the keyloggers used in the evaluation and summarizes the detection results. All the keyloggers were detected without generating any false positives. For the last two keyloggers in the list, we were not able to provide any detection result since no consistent log file was ever generated in the two cases even after repeated experiments[1]. In every other case, our detection system was able to detect the keylogger correctly within a few seconds.

Virtuoza Free Keylogger required a longer window of observation to be detected. The *Virtuoza Free Keylogger* was indeed the only keylogger to use some form of aggressive buffering, with keystrokes stored in memory and flushed out to disk at regular time intervals. Nevertheless, we were still able to collect consistent samples from buffer flush events and report high values of PCC with the normalized version of the input pattern.

In a few other cases, keystrokes were kept in memory but flushed out to disk as soon as the keylogger detected a change of focus. This was the case for *Actual Keylogger*, *Revealer Keylogger Free*, and *Refog Keylogger Free*. To deal with this common buffering strategy efficiently, our detection system enforces a change of focus every time a sample is injected into the system. Other buffering strategies and possible evasion techniques are discussed in detail in Sec. 5.

Furthermore, some of the keyloggers examined included support for encryption and most of them used variable-length encoding to store special keys. As Sec. 4.2 demonstrates with experimental results, our algorithm can deal with these nuisances transparently with no effect on the resulting PCC measured.

Another potential issue arises from most keyloggers dumping a fixed-format header on the disk every time a change of focus is detected. The header typically contains the date and the name of the target application. Nonetheless, as we designed our detection system to change focus at every sample, the header is flushed out to disk at each time interval along with all the keystrokes injected. As a result, the output pattern monitored is simply a location transformation of the original, with a shift given by size of the header itself. Thanks to the location invariance property, our detection algorithm is naturally resilient to this transformation, regardless of the particular header size used.

[1] Both keyloggers were installed on Windows XP SP3 and instructed to output their log in a specific directory. However, since no logs have been subsequently produced, we assumed they were not fully compatible with the underlying environment.

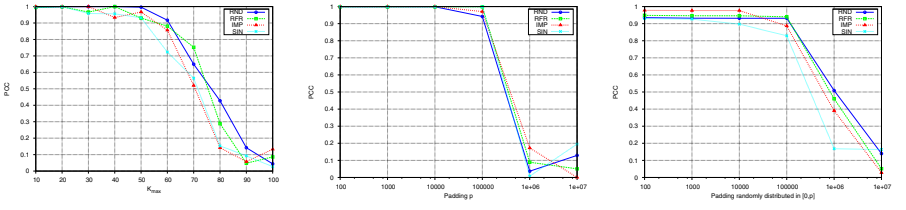
4.2 False Negatives

In our approach, false negatives may occur when the output pattern of a keylogger scores an unexpectedly low PCC value. To test the robustness of our approach with respect to false negatives, we made several experiments with our own artificial keylogger. In its basic version, our prototype keylogger merely logs each keystroke on a text file on the disk.

Our evaluation starts with the impact of the maximum number of keystrokes per time interval K_{max}. High K_{max} values are expected to increase the level of variability, reduce the amount of noise, and reproduce a more distinct distribution in the output stream of a keylogger. Nevertheless, the keystroke rate is clearly bound by the size of the time interval T. Figure 2(a) depicts this scenario with $N = 50$ and $T = 1000$ ms. For each pattern generation algorithm, we plot the PCC measured with our prototype keylogger. This graph shows very high values of PCC for $K_{max} < 50$. For $K_{max} > 50$, regardless of the pattern generation algorithm, the PCC linearly decreases. The effect observed is due to the inability of the system to absorb more than $K_{max} \approx 50$ in the given time interval. We observe analogous results whether we plot the PCC against different values of T. Our results (hereby not reported) shows that the PCC value becomes steady for $T \geq 150$.

We conducted further experiments to analyze the impact of the number of samples N. As expected, the PCC is steady regardless of the value of N. This behavior should not suggest, however, that N has no effect on the production of false negatives. When noise in the output stream is to be expected, higher values of N are indeed desirable to produce more accurate measures of the PCC and avoid potential false negatives.

We have also simulated the effect of several possible input-output transformations. First, we experimented with a keylogger using a non-trivial fixed-length encoding for keystrokes. Figure 2(b) depicts the results for different values of padding $100 < p < 10000000$. A value of $p = 100$ simulates a keylogger writing 100 bytes on the disk for each eavesdropped keystroke. As discussed in Sec. 3.4, the PCC should be unaffected in this case and presumably exhibit a constant behavior. The graph confirms this basic intuition, but shows the PCC dropping linearly after around $p = 100000$ bytes. This behavior is due to the limited I/O throughput that can be achieved within a single time interval. Let us now



(a) PCC in function of K_{max}. (b) Effect of constant padding. (c) Effect of random padding.

Fig. 2. The effect of the parameters on the PCC measured with our keylogger

consider a scenario where the keylogger writes a random amount of characters r, with $0 \leq r \leq p$, for each eavesdropped keystroke. This is interesting to evaluate the impact of several different conditions. First, the experiment simulates a keylogger randomly dropping keystrokes with a certain probability. Second, the experiment simulates a keylogger encoding a number of keystrokes with special sequences, e.g. `CTRL` logged as `[Ctrl]`. Finally, this scenario investigates the impact of a keylogger performing variable-length encryption or other variable-length transformations. Results for different values of p are depicted in Fig. 2(c). As observed in Fig. 2(b), the PCC only drops at saturation. The graph still reveals a steady behavior with the stable value of the PCC only slightly affected (PCC ≈ 0.95), despite the extreme level of noise introduced. Experiments with non-uniform distributions of r in the same interval yield similar results. We believe these results are very encouraging to demonstrate the strength of our detection technique with to respect to false negatives, even in presence of severe data transformations.

4.3 False Positives

In our approach, false positives may occur when the output pattern of some benign process accidentally scores a significant PCC value. If the value happens to be greater than the particular threshold selected, a false detection is triggered. In this section, we evaluate our prototype system to understand how often such circumstances may occur in practice.

To generate representative synthetic workloads for the PC user, we relied on the widely-used SYSmark 2004 SE suite [4]. The suite leverages common Windows interactive applications[2] to generate realistic workloads that mimic common user scenarios with input and think time. In its 2004 SE version, SYSmark supports two individual workload scenarios: Internet Content Creation (Internet workload from now on), and Office Productivity (Office workload from now on). In addition to the workload scenarios supported by SYSmark, we have also experimented with another workload that simulates an idle Windows system with common user applications[3] running in the background. In the Idle workload scenario, we allow no user input and focus on the I/O behavior of a number of typical background processes.

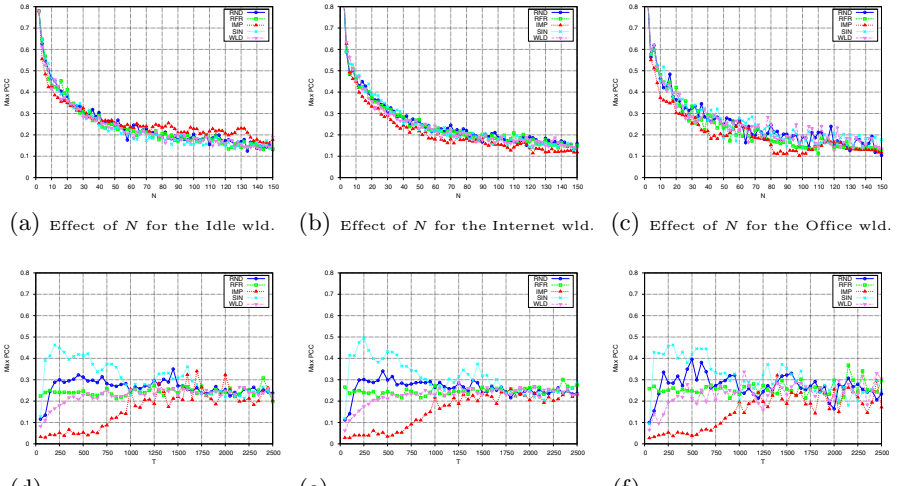
For each scenario, we repeatedly reproduced the synthetic workload on a number of machines and collected I/O traces of all the running processes for several possible sampling intervals T. Each trace was stored as a set of output patterns and broken down into k consecutive chunks with N samples. Every experiment was repeated over $k/2$ rounds, once for each pair of consecutive chunks. At each round, the output patterns from the first chunk were used to train our workload-aware pattern generation algorithm, while the second chunk was used for testing.

[2] The set of user programs is available at the following web site `http://www.bapco.com/products/sysmark2004se/applications.php`.

[3] Skype 4.1, Pidgin 2.6.3, Dropbox 0.6.556, Firefox 3.5.7, Google Chrome 5.0.307, Avira Antivir Personal 9.0, Comodo Firewall 3.13, and VideoLAN 1.0.5.

In the testing phase, we measured the maximum absolute PCC between every generated input pattern of length N and every output pattern in the testing set. At the end of each experiment, we averaged all the results. We tested all the workload-agnostic and workload-aware pattern generation algorithms introduced earlier.

We start with an analysis of the pattern length N, evaluating its effect while fixing T to 1000 ms. Similar results can be obtained with other values of T. Figures 3(a), 3(b), 3(c) depict the results of the experiments for the Idle, Internet, and Office workload. As aforementioned, the behavior observed is very similar in all the workload scenarios examined. The only noticeable difference is that the Office workload presents a slightly more unstable PCC distribution. This is probably due to the more irregular I/O workload monitored.



(a) Effect of N for the Idle wld. (b) Effect of N for the Internet wld. (c) Effect of N for the Office wld.

(d) Effect of T for the Idle wld. (e) Effect of T for the Internet wld. (f) Effect of T for the Office wld.

Fig. 3. The effect of the parameters and the workload on the maximum PCC measured with regular system processes.

As shown in the graphs, the maximum PCC value decreases exponentially as N increases. This confirms the intuition that for small N, the PCC may yield unstable and inaccurate results, possibly assigning very high correlation values to regular system processes. For example, using input patterns of length $N < 10$ typically results in misleading PCC values in the range of 0.5-0.8 for one or more regular system process. Fortunately, the maximum PCC decreases very rapidly and, for example, for $N > 30$, its value is constantly below 0.35. As far as the pattern generation algorithms are concerned, they all behave very similarly. Notably, RFR yields the most stable PCC distribution. This is especially evident for the Office workload. In addition, our workload-aware algorithm WLD does not perform significantly better than any other workload-agnostic pattern generation algorithm. This strongly suggests that, independently of the value of

N, the output pattern of a process at any given time is not in general a good predictor of the output pattern that will be monitored next. This observation generally reflects the low level of predictability in the I/O behavior of a process.

From Figures 3(d), 3(e), 3(f) we can observe the effect of the parameter T on input patterns generated by the IMP algorithm. The experiments shown here have been conducted by fixing the pattern length to an arbitrary stable value of $N = 50$. For small values of T, IMP constantly outperforms all the other algorithms by producing extremely anomalous I/O patterns for the given T in any workload scenario. As T increases, the irregularity becomes less evident and IMP matches more closely the behavior of the other algorithms.

In general, for reasonable values of T, all the pattern generation algorithms reveal a similar and constant distribution of the PCC. This confirms the property of self-similarity of the I/O traffic. As expected, the PCC measured is generally independent of the interval T. Notably, RFR and WLD reveal a more steady distribution of the PCC. This is probably due to the use of a fixed range of values in both algorithms. This also confirms the intuition that more variability in the input pattern leads to more accurate and stable results.

For very small values of T, we also note that WLD performs significantly better than the average. This is a hint that predicting the I/O behavior of a generic process in a fairly accurate way is only realistic for small windows of observation. In all the other cases, we believe that the complexity of implementing a workload-aware algorithm largely outweighs its benefits. For small values of T, we also found the SIN algorithm to be more prone to generation of false positives. In our analysis, we found that similar PCC distributions can be obtained with very different types of workload. This suggests that it is possible to select the same threshold for many different settings. For reasonable values of N and T, we found that a threshold of $0.5 - 0.6$ is usually sufficient to rule out the possibility of false positives, while being able to detect most keyloggers effectively. In addition, the use of a stable pattern generation algorithm like RFR could also help minimize the level of unpredictability across many different settings.

5 Evasion Techniques

Despite we were able to detect all the existing keyloggers, there are some evasion techniques that keyloggers may employ to make detection more difficult. For instance, a keylogger may rely on some kind of aggressive buffering, namely flushing its buffer every 12 hours. In this case, since we would need 12 hours to collect a single sample, increasing the amount of samples is not desired solution. We point out that the model underlying our detection technique is not accountable for this limitation. In fact, monitoring the memory accesses of the running processes would promptly make the detection process immune to such behavior. However, since monitoring the memory accesses is not available by means of unprivileged APIs, we reckon that such benefits are mitigated by the need of running the OS in a virtualized environment. A more complex behavior is a keylogger actively performing I/O activities. Although this class of keylogger is

hypothetical, our model can easily be augmented to handle this type of keyloggers. The solution is to inject rates of keystrokes higher than the I/O generated by the disguisement activities. In this scenario the component able to inject most of the keystrokes would make its pattern to emerge. Further research is advised to assess the viability of such a countermeasure against this evasion technique.

6 Related Work

Despite our approach is the first and only technique to solely rely on unprivileged execution environments, several works recently dealt with the detection of privacy-breaching malware.

The technique of detecting malware by means of modeling their behavior has been previously proposed by Kirda et al. [12]. Their approach is tailored to detect malware running as Internet Explorer loadable modules. Modules both monitoring the user's activity and disclosing such data to other processes are flagged as malware. Their analysis in fact defines a malware behavior in terms of API calls invoked in response to browser events. However, as we previously discussed, the API calls a keylogger leverages are commonly used by legitimate components. Their approach is therefore prone to false positives that only a continuously updated white-list may be able to counter.

Slightly more sophisticated approaches are the ones detecting when known APIs are exploited. Since user-space keyloggers are known to target a little set of APIs, this approach perfectly fits our case. Aslam et al. [3] adopt the approach to disassemble executable in order to look for the mentioned API calls. Unfortunately, all these calls are commonly used by legitimate applications; detecting keyloggers by such means would produce a remarkable amount of false positives. Xu et al. [20] push this technique a little further. They interpose a function between the API and any program calling it; this function denies the delivering of the keystroke to the keylogger by means of altering its type (from `WM_KEYDOWN` to `WM_CHAR`). However, since they rely on the ability to interpose the function before the keylogger, a malware aware of this countermeasure can easily elude it.

A step a little closer to our approach is discussed by AlHammadi et al. in [1]. Their approach defines a malware behavior in terms of the invoked API functions. To be more precise, they collect the frequency of API calls invoked to (i) intercept keystrokes, (ii) writing to a file, and (iii) sending bytes over the network. A malware is then flagged as such whether two frequencies are found to be highly correlated. Since no bogus events are sent to the system (no injection of crafted input), the correlation may be not be as strong as expected. The correlation value would be even more impaired in case of any delay introduced by the malware. Moreover, since the whole analysis is focused on a specific bot, it lacks a proper discussion on both false positive and false negatives of their quantitative analysis. In our approach we focus on the actual written bytes and consequently adopt a different correlation metric, i.e. PCC, that instead is linear. Due to its linearity, any data transformation (such as encryption) would not help the malware in evading our detection. Our approach is also immune to malware reasonably

buffering the collected data: as long as the minimum rate of injected keystrokes flushes the buffer in question, our approach preserves its effectiveness. In case such a requirement can not be met, our technique can be easily extended by means of aggregating consecutive samples as we explain in Sec. 5.

A similar technique comprising of both quantitative analysis and injection routine is sketched by Han et al. in [10]. However, besides being a privileged approach like [1] and [6], it merely relies on the amount of API calls triggered in response to a certain amount of keystrokes. However, the assumption that a certain amount of keystrokes implies a fixed amount of API calls is not always true. It is how a program is implemented that determines in how many chunks a stream of data is written to disk. In our approach we rely on more precise measurements that are also available by means of unprivileged APIs, namely the amount of bytes a process writes.

In conclusion, notable approaches recently attempted to generalize the behavior deemed malicious. In particular, in [16,7] the authors attempt to identify trigger-based behavior by means of mixing concrete and symbolic execution. In such a way they aim to explore all the possible execution paths that a malware may reproduce during execution. As the authors in [16] admit, however, automating the detection of trigger-based behavior is an extremely challenging task requiring advanced privileged tools. The problem is also undecidable in the general case.

7 Conclusions

In this paper we presented an unprivileged black-box approach for accurate detection of the most common keyloggers, i.e. user-space keyloggers. We modeled the behavior of a keylogger by means of correlating the input, i.e. the keystrokes, to the I/O pattern produced by the keylogger. Moreover, since the input to the system was known, we augmented our model by introducing the ability to artificially inject keystrokes. We then discussed the problem of choosing the best input pattern to improve our detection rate. Subsequently we implemented our architecture on the operating system most vulnerable to the threat of keyloggers, i.e. Windows. We also gave implementation details to accommodate different operating systems, thus obtaining an OS independent architecture. We then tested the prototype against a real case scenario; the results met our expectations: given a proper threshold, we were able to detect 100% of the most common free keyloggers [15] completely avoiding any false positive.

As future works, we will investigate the tradeoff of giving up the constraint of an unprivileged execution environment; accessing privileged APIs would allow our detector to monitor memory accesses of running processes.

References

1. Al-Hammadi, Y., Aickelin, U.: Detecting bots based on keylogging activities. In: Proceedings of the Third International Conference on Availability, Reliability and Security, pp. 896–902 (2008)

2. Aldrich, J.: Correlations genuine and spurious in pearson and yule. Statistical Science 10(4), 364–376 (1995)
3. Aslam, M., Idrees, R., Baig, M., Arshad, M.: Anti-Hook Shield against the Software Key Loggers. In: Proceedings of the 2004 National Conference on Emerging Technologies, p. 189 (2004)
4. BAPCO: SYSmark 2004 SE (2004), http://www.bapco.com/products/sysmark2004se/
5. Benesty, J., Chen, J., Huang, Y.: On the importance of the pearson correlation coefficient in noise reduction. IEEE Transactions on Audio, Speech, and Language Processing 16(4), 757–765 (2008)
6. Borders, K., Zhao, X., Prakash, A.: Siren: Catching evasive malware (short paper). In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 76–85 (2006)
7. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. Advances in Information Security 36, 65–88 (2008)
8. Goodwin, L., Leech, N.: Understanding correlation: Factors that affect the size of r. The Journal of Experimental Education 74(3), 249–266 (2006)
9. Grebennikov, N.: Keyloggers: How they work and how to detect them, http://www.viruslist.com/en/analysis?pubid=204791931
10. Han, J., Kwon, J., Lee, H.: Honeyid: Unveiling hidden spywares by generating bogus events. In: Proceedings of The Ifip Tc 11 23rd International Information Security Conference, pp. 669–673 (2008)
11. Hsu, W., Smith, A.: Characteristics of I/O traffic in personal computer and server workloads. IBM System Journal 42(2), 347–372 (2003)
12. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based spyware detection. In: Proceedings of the 15th USENIX Security Symposium (USENIX Security 2006) (2006)
13. Kochenberger, G., Glover, F., Alidaee, B.: An effective approach for solving the binary assignment problem with side constraints. Internation Journal of Information Technology and Decision Making 1, 121–129 (2002)
14. Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistics Quarterly 2, 83–97 (1955)
15. Security Technology Ltd.: Testing and reviews of keyloggers, monitoring products and spy software (spyware) (2009), http://www.keylogger.org/monitoring-free-software-review/
16. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Proceeding of the 28th IEEE Symposium on Security and Privacy (SP 2007), pp. 231–245 (May 2007)
17. San Jose Mercury News: Kinkois spyware case highlights risk of public internet terminals (2009), http://www.siliconvalley.com/mld/siliconvalley/news/6359407.htm
18. Rodgers, J.L., Nicewander, W.A.: Thirteen ways to look at the correlation coefficient. The American Statistician 42(1), 59–66 (1988)
19. Strahija, N.: Student charged after college computers hacked (2003), http://www.xatrix.org/article2641.html
20. Xu, M., Salami, B., Obimbo, C.: How to protect personal information against keyloggers. In: Proceedings of the 9th International Conference on Internet and Multimedia Systems and Applications, IASTED 2005 (2005)

Generating Client Workloads and High-Fidelity Network Traffic for Controllable, Repeatable Experiments in Computer Security*

Charles V. Wright, Christopher Connelly, Timothy Braje,
Jesse C. Rabek**, Lee M. Rossey, and Robert K. Cunningham

Information Systems Technology Group
MIT Lincoln Laboratory
Lexington, MA 02420
{cvwright,connelly,tbraje,lee,rkc}@ll.mit.edu, jesrab@alum.mit.edu

Abstract. Rigorous scientific experimentation in system and network security remains an elusive goal. Recent work has outlined three basic requirements for experiments, namely that hypotheses must be *falsifiable*, experiments must be *controllable*, and experiments must be *repeatable* and *reproducible*. Despite their simplicity, these goals are difficult to achieve, especially when dealing with client-side threats and defenses, where often user input is required as part of the experiment. In this paper, we present techniques for making experiments involving security and client-side desktop applications like web browsers, PDF readers, or host-based firewalls or intrusion detection systems more *controllable* and more easily *repeatable*. First, we present techniques for using statistical models of user behavior to drive real, binary, GUI-enabled application programs in place of a human user. Second, we present techniques based on adaptive replay of application dialog that allow us to quickly and efficiently reproduce reasonable mock-ups of remotely-hosted applications to give the illusion of Internet connectedness on an isolated testbed. We demonstrate the utility of these techniques in an example experiment comparing the system resource consumption of a Windows machine running anti-virus protection versus an unprotected system.

Keywords: Network Testbeds, Assessment and Benchmarking, Traffic Generation.

1 Introduction

The goal of conducting disciplined, reproducible, "bench style" laboratory research in system and network security has been widely acknowledged [1,2], but remains difficult to achieve. In particular, Peisert and Bishop [2] outline three

* This work was supported by the US Air Force under Air Force contract FA8721-05-C-0002. The opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

** Work performed as a student at MIT. The author is now with Palm, Inc.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 218–237, 2010.

basic requirements for performing good experiments in security: (*i*) Hypotheses must be *falsifiable*—that is, it must be possible to design an experiment to either support or refute the hypothesis. Therefore, the hypothesis must pertain to properties that are both *observable* and *measurable*. (*ii*) Experiments must be *controllable*; the experimenter should be able to change only one variable at a time and measure the change in results. (*iii*) Finally, experiments should be both *repeatable*, meaning that the researcher can perform them several times and get similar results, and *reproducible*, meaning that others can recreate the experiment and obtain similar results.

Unfortunately, designing an experiment in system or network security that meets these requirements remains challenging. Current practices for measuring security properties have recently been described as "ad-hoc," "subjective," or "procedural" [3]. Experiments that deal primarily with hardware and software may be extremely controllable, and recent work [4,5,6,7,8] has explored techniques for deploying and configuring entire networks of servers, PCs, and networking equipment on isolated testbeds, disconnected from the Internet or other networks, where malicious code may safely be allowed to run with low risk of infecting the rest of the world. However, the recent shift in attacks from the server side to the client side [9,10,11,12] means that an experiment involving any one of many current threats, such as drive-by downloads [13] cross-site scripting, or techniques for detecting and mitigating such intrusions, must account for the behavior of not only the hardware and software of the computing infrastructure itself, but also the behavior of the human users of this infrastructure. Humans remain notoriously difficult to control, and experiments using human subjects are often expensive, time consuming, and may require extensive interaction with internal review boards.

Repeatability of experiments on the Internet is difficult due to the global network's scale and its constant state of change and evolution. Even on an isolated testbed, repeatability is hampered by the sheer complexity of modern computer systems. Even relatively simple components like hard disks and Ethernet switches maintain state internally (in cache buffers and ARP tables, respectively), and many components perform differently under varying environmental conditions e.g. temperature. Many recent CPUs dynamically adjust their clock frequency in reaction to changes in temperature, and studies by Google suggest that temperature plays an important role in the failure rates of hard disk drives [14]. Reproducibility is even harder. It is unclear what level of detail is sufficient for describing the hardware and software used in a test, but current practices in the community likely fall short of the standards for publication in the physical sciences.

The contributions of this paper address two of the above requirements for performing scientific experiments in security. Specifically, we describe techniques that enable *controllable*, *repeatable* experiments with client-side attacks and defenses on isolated testbed networks. First, we present techniques for using statistical models of human behavior to drive real, binary, GUI-enabled application programs running on client machines on the testbed, so that tests can be

performed without the randomness or privacy concerns inherent to using human subjects. Second, we present adaptive replay techniques for producing convincing facsimiles of remotely-hosted applications (e.g. those on the World Wide Web) that cannot themselves be installed in an isolated testbed network, so that the client-side applications have something to talk to. In doing so, we generate workloads on the hosts and traffic on the network that are both highly controllable and repeatable in a laboratory testbed setting.

On the client side, our approach is to construct a Markov chain model for the way real users interact with each application. Then, during the experiment, we use the Markov chains to generate new event streams similar in distribution to those generated by the real users, and use these to drive the applications on the testbed. This provides a realistic model of measured human behavior, offers variability from trial to trial, and provides an experimenter with the ability to change model parameters to explore new user classes. It also generates a reasonably realistic set of workloads on the host, in terms of running processes, files and directories accessed, open network ports, system call sequences, and system resource consumption (e.g. CPU, memory, disk). Many of these properties of the system are important for experiments involving defensive tools like firewalls, virus scanners, or other intrusion detection systems because they are used by such systems to detect or prevent malicious behavior. Furthermore, because we run unmodified application program binaries on the testbed hosts, we can closely replicate the attack surface of a real network and use the testbed to judge the effectiveness of various real attacks and defenses against one another.

Using real applications also allows us to generate valid traffic on the testbed network, even for complicated protocols that are proprietary, undocumented, or otherwise poorly understood. We discuss related work in more detail in the following section, but for now it suffices to say that almost all existing work on synthetically generating network traffic focuses on achieving realism at only one or two layers of the protocol stack. In contrast, our approach provides realistic traffic all the way from the link layer up to and including the contents of the application layer sessions.

For example, by emulating a user replying to an email, with just a few mouse click events, we can generate valid application-layer traffic in open protocols like DNS, IMAP, and LDAP, proprietary protocols including SMB/CIFS, DCOM, and MAPI/RPC (Exchange mail). This is, of course, in addition to the SMTP connection used to send the actual message. Each of these connections will exhibit the correct TCP dynamics for the given operating system and will generate the proper set of interactions at lower layers of the stack, including DNS lookups, ARP requests, and possibly Ethernet collisions and exponential backoff. Moreover, if a message in the user's inbox contains an exploit for his mail client (like the mass-mailing viruses of the late 1990s and early 2000s), simply injecting a mouse click event to open the mail client may launch a wave of infections across the testbed network.

For the case where the actual applications cannot be installed on the isolated test network, we present techniques based on adaptive replay of application

dialog that allow us to quickly and efficiently reproduce reasonable mock-ups that make it appear across the network as if the real applications were actually running on the testbed. These techniques are particularly useful for creating a superficially realistic version of the modern World Wide Web, giving the illusion of connectedness on an isolated network.

To illustrate the utility of these techniques, we perform a simple experiment that would be labor intensive and time consuming to conduct without such tools. Specifically, we investigate the performance impact of open source anti-virus (AV) software on client machines. Conventional folk wisdom in the security community has been that AV products incur a significant performance penalty, and this has been used to explain the difficulty of convincing end users to employ such protection. Surprisingly, relatively little effort has been put in to quantifying the drop in performance incurred, perhaps due to the difficulty of performing such a test in a controllable and repeatable manner.

The remainder of the paper is organized as follows. In Section 2, we review related work in network testbeds, automation of GUI applications, modeling user behavior, and network traffic generation. In Section 3, we present our techniques for driving real binary applications and for crafting reasonable facsimiles of networked applications that we cannot actually install on the testbed. In Section 4, we walk through a simple experiment to demonstrate the utility of these techniques and to highlight some challenges in obtaining repeatable results. Finally, we conclude in Section 5 with some thoughts on future directions for research in this area.

2 Related Work

Several approaches for configuring, automating, and managing network laboratory testbeds have recently been proposed, including Emulab [4], FlexLab [5], ModelNet [6], and VINI [7]. Our group's LARIAT testbed platform [8] grew out of earlier work in the DARPA intrusion detection evaluations [15,16] and was designed specifically for tests of network security applications. More recently, along with others in our group, two of the current authors developed a graphical user interface for testbed management and situational awareness [17] for use with LARIAT. The DETER testbed [18] is built on Emulab [4] and, like LARIAT, is also geared toward network security experiments. The primary contribution of this paper, which is complementary to the above approaches, is to generate client-side workloads and network traffic for experiments on such testbeds. The techniques in Section 3.1 were first described in the fourth author's (unpublished) MIT Master's thesis [19]. USim, by Garg et al. [20], uses similar techniques for building profiles of user behavior, and uses scripted templates to generate data sets for testing intrusion detection systems.

Our server-side approach for emulating the Web is similar to the dynamic application layer replay techniques of Cui et al. [21,22] and Small et al. [23]. Like our client-side approach, the MITRE HoneyClient [24] and Strider HoneyMonkeys from Microsoft Research [25] drive real GUI applications, but that work

focuses narrowly on automating web browsers to discover new vulnerabilities and does not attempt to model the behavior of a real human at the controls. Software frameworks exist for the general-purpose automation of GUI applications, including `autopy` [26] and SIKULI [27], but these also require higher-level logic for deciding which commands to inject. PLUM [28] is a system for learning models of user behavior from an instrumented desktop environment. Simpson et al. [29] and Kurz et al. [30] present techniques for deriving empirical models of user behavior from network logs.

There is a large body of existing work on generating network traffic for use on testbeds or in simulations, but unfortunately most of these techniques were not designed for security experiments. Simply replaying real traffic [31,32] does not allow for controllable experiments. Other techniques for generating synthetic traffic based on models learned from real traffic [33,34,35,36,37] can match several important statistical properties of the input trace at the Network and Transport layers. However, because these approaches do not generate application layer traffic, they are not compatible with many security tools like content-based filters and intrusion detection or prevention systems, and they cannot interact with real applications on a testbed. Sommers et al. [38] present a hybrid replay-synthesis approach that may be more appropriate for some experiments in security. Mutz et al. [39], Kayacik and Zincir-Heywood [40], and other work by Sommers et al. [41] generate traffic specifically for the evaluation of defensive tools.

Commercial products from companies including Ixia, BreakingPoint, and Spirent can generate application-layer traffic, but their focus is on achieving high data rates rather than realistic models of individual user behavior, and their implementations do not necessarily exhibit the same attack surface as the real applications.

3 Traffic and Workload Generation Techniques

Although our techniques could potentially be applied using any of the existing network testbeds [4,5,6,7,8,18], our current implementation is built as an extension of own testbed platform, LARIAT [8], which provides a centralized database for experiment configuration and logging and a graphical user interface for launching automated tasks to configure the testbed and for controlling and monitoring experiments. Since the publication of [8], the scope of the project has expanded significantly. LARIAT has been used to run distributed experiments on testbeds of more than a thousand hosts. In addition to the user model-driven actuation capabilities and internet reproduction described in this paper, components have been added for automatically configuring client and server software, controlling hosts across a testbed, visualizing the configuration and logged data [17], and for distributing control across remote physical locations. The current version can drive user-model behavior on a number of different operating system and physical device platforms including smart phones and router consoles.

3.1 Client-Side Workload Generation

Our approach is to emulate a human user by injecting input events to applications via the operating system. In principle, we could use any number of possible techniques to determine what events to inject and when. One simple approach would be to simply record the sequence of events generated by a real user, and replay them verbatim to the applications on the testbed. While this "capture/replay" approach offers a level of realism that is difficult to match with synthetic workloads, it fails the requirement that experiments be *controllable*.

Our techniques strike a careful balance between realism of the workloads and controllability of the experiment. We record the inputs generated by real human users and then train a hierarchical Markov chain model for the events sent to each application. Then, during the experiment, we simulate from the Markov chains to generate new event streams similar in distribution to those generated by the real users, and use these to drive the applications on the testbed.

Application User State Machines. We call these models Application User State Machines, or AUSMs, because the Markov chain models describe a finite state machine model of a human user of the application. Formally, an AUSM is defined as a 4-tuple (n, A, M, X), where n is the number of states in the finite state machine model, $A = \{a_{ij} : i, j < n\}$ is the Markov chain state transition matrix, $M = \{m_i : i < n\}$ is a set of second-level models for the outputs produced by each state, and $X = \{X_{ij} : i, j < n\}$ is a set of models describing the interarrival time distribution when an event of type i is immediately followed by an event of type j. We describe the training and event generation processes for these models in greater detail in the following paragraphs.

Setting AUSM Parameters. To collect training data for the AUSM's, we use the DETOURS framework [42] from Microsoft Research to instrument a set of Windows desktop machines as they are driven by real human users. During the training interval, we record the event ID, process ID, and arrival time of each COM (Component Object Model) event on these instrumented systems for some length of time. We then use the sub-sequence of events corresponding to each application to set the parameters for a hierarchical Markov chain model that we then use to drive the given application on the testbed.

To create an AUSM, we begin by creating one state for each event ID. We count the number of times in the training data where event i was immediately followed by event j, and store this count as c_{ij}. We then compute the probability of a transition from state i to state j, and store this in the Markov model's state transition matrix as:

$$a_{ij} = \frac{c_{ij}}{\sum_k c_{ik}}$$

Modeling State Output Distributions. To allow for flexibility in the level of detail provided by the AUSMs, the outputs of each state are represented using a second level of models. Some states may always produce the same output, e.g. a state that generates a mouse click on the "Start" button. Others, like the state that

generates input for a text box in Internet Explorer, or the word processor input model, use an n-gram word model of English to produce blocks of text at a time.

If we have no other source of data, these output models can be trained using the values observed during the training data collection. In other cases, where we have some expert knowledge of the application, the output models can be trained using other, larger external data sources. For example, the model that generates text for the body of an email could be trained using the contents of real emails in the Enron corpus [43]. In our experiments, we use a locally-collected corpus of real emails from the authors' inboxes to train a bigram word model of English text.

Modeling Event Interarrival Times. Each state transition edge (i, j) in the AUSM also has an associated interarrival time distribution X_{ij}, which characterizes the delay between events when event i is immediately followed by event j. Typically, waiting time distributions are well described by the exponential distribution (e.g. time between buses arriving at a bus stop, time between major hurricanes, etc.). However, the data collected from our users' workstations exhibits a heavier tail than the exponential distribution, with more wait times that are much longer than the mean. Some so-called "heavy-tailed" distributions that occur as a result of user interaction have been shown to be well described by a power-law or Pareto distribution in the past [44], although the Pareto distribution also does not appear to be a good fit for our event interarrivals. Figure 1 shows the observed empirical distribution of COM event interarrival times for one state transition, together with the best-fit exponential and Pareto distributions.

Our hypothesis for the poor fit of these two distributions is that there are actually two sub-populations of event interarrival times. In the first case, the user is actively engaged with the application, generating events at shorter and more regular intervals. In the second case, the user may switch to another application or disengage from the system entirely to perform some other task, such

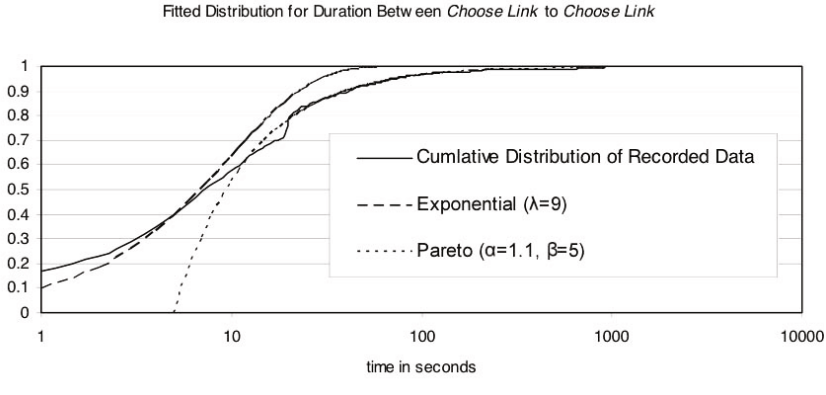


Fig. 1. Empirical distribution of event interarrival times, with best-fit Exponential and Pareto distributions

as answering the telephone, reading a paper, going to a meeting, going home for the night, or even going on vacation while leaving the system up and running. To capture this bimodal distribution, we use a mixture model with one Exponential component to represent the periods of active engagement and one Pareto component to represent the longer periods of inactivity.

Generating Client Workloads. In this section we explain how the state machine models developed above can be used to feed input to application programs on a client machine to generate workloads on the host and traffic on the testbed network. Figure 2 shows at a high level how our modules interface with the Windows OS and applications on the client-side system under test (SUT) to achieve the illusion of a human user at the controls.

Regarding Repeatability. We note that, in order to achieve repeatable experimental results, the entire testbed needs to be started from a fixed state at the beginning of each run. While we believe the approach we describe here is a necessary condition for obtaining repeatable experimental results, this alone is not sufficient. We elaborate on other techniques for improving the repeatability of an example experiment in Section 4.

To enable repeatable outputs from our state machines, we store a master random seed in the LARIAT database for each experiment. As part of setting up the testbed for the experiment, each host generates its own unique random seed as a hash of the master random seed and a unique host identifier assigned to it by the LARIAT testbed management system. At the beginning of an experiment, each host instantiates a Mersenne Twister [45] pseudo-random number generator, seeded with its host seed. This PRNG is then used to drive the state machines as explained above. Thus, by keeping the master seed unchanged for several runs of the experiment, we can repeat a test many times and get the same sequence of actions from the state machines in each run. Conversely, by varying the master seed, we can explore the space of possible user actions and the corresponding experimental outcomes.

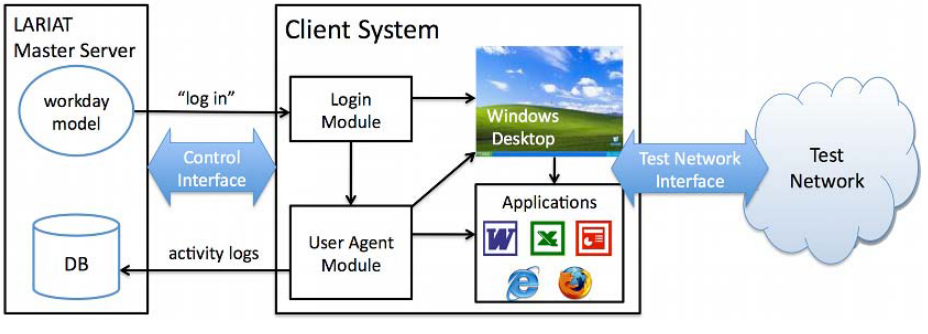


Fig. 2. Client-side traffic generation overview

To simulate the user arriving at the machine and logging in, the master LARIAT server sends a message to the client host's login module over the control interface, instructing it to log in the given user. On Windows NT, 2000, and XP systems, the login module is implemented as a GINA, a dynamic-link library used by the Windows *Winlogon* process for Graphical Identification and Authentication [46]. On Windows Vista and newer versions, it runs as a service. In either case, the module provides login credentials to the OS to start up a desktop session for the given user. It also launches the user agent module, which generates user input to drive the Windows desktop and applications from that point forward.

Upon login, the user agent module starts with a pseudorandomly-selected AUSM and, if necessary, launches the corresponding application. Then, until the user agent process receives a signal instructing it to log the user out, it generates input for the applications by driving the state machines as follows.

In state i, the user agent first samples from state i's output model m_i to generate an input to the application. It injects the input events using the Microsoft COM APIs or as keyboard events so that, from the applications' point of view, these events are delivered by the operating system just as if they had been generated by a real human user. Then, the user agent selects the next state j by pseudorandomly sampling from row i of the Markov model's state transition matrix A. The user agent samples a pseudorandom delay x from the AUSM's event interarrival time distribution X_{ij}. It then sleeps for x seconds, resets the current state to j, and repeats the process. In some cases, the output of state j may be to launch a new application or switch to another running application. In such cases, the user agent also switches to using the new application's AUSM in the next iteration.

3.2 Server Side Techniques

For our client-side workload generation techniques to truly be useful on an isolated testbed network, there must be something for the client side applications to talk to. Sometimes this is relatively straightforward. For example, simply installing and configuring a Microsoft Exchange email and calendaring server on the client's local area network is mostly sufficient to enable the MS Outlook AUSM to function normally. Our previous work [8] presents techniques for generating emails for the virtual users to receive, and of course the Domain Name Service and IP layer routing must be properly configured on the testbed so that clients can discover one another's SMTP servers and transmit the actual mail. Some testbed management systems [47,8,18] handle part or all of this setup process.

For some other network applications, most notably the world-wide web, setting up a realistic environment on an isolated network is much more challenging. Although installing a server for the underlying HTTP protocol is not especially difficult, getting realistic content is. In the early days of the web, most pages consisted solely of static content, which could easily be downloaded and "mirrored" on another server to easily replicate the page. While some web pages

still use this model, for example many researchers' profile pages, the majority of the most popular web sites are currently powered by special-purpose, proprietary programs that dynamically generate page content and are only accessible as a service. Some web applications for dynamically generating page content are available for installation on the testbed, either as software packages, or as a hardware appliance such as the Google Search Appliance [48], and we do make use of several such products, including the open source *osCommerce* [49] e-commerce engine, the *GreyMatter* weblog software, and Microsoft Exchange's webmail interface.

However, to make it appear on the surface as if the isolated testbed network is actually connected to the Internet, more sophisticated techniques are required. Our approach is to use dynamic application-layer replay techniques like those developed by Cui et al. [22,21] and Small et al. [23] for creating lightweight server-side honeypots. We elaborate on our approach in the following sections.

Collecting Data. We begin by downloading a large number of web pages using what is essentially a client-side honeypot [25,24]. That is, we run a web browser (in our case Microsoft Internet Explorer) on a Windows operating system in a virtual machine, and we script it to automatically download a list of URLs via a consumer-grade cable modem connection to the Internet. For each URL in the list, we retrieve the page using the honeyclient and record the full contents of each packet that is generated. We revert the VM to a clean state after each page retrieval. For broad coverage of the Web, we begin with a list of over ten thousand URLs from a categorized directory of links such as Mozilla's Open Directory Project [50] or Yahoo!'s directory [51], as well as lists of the most popular URLs from the Alexa.com rankings [52]. For increased realism, we can script the honey client to "crawl" more links to provide increased depth for a given interactive site.

Then, we perform TCP stream reassembly on the captured packets to reconstruct the application-layer conversations for each URL and all of the associated HTTP sessions. For each HTTP request in the collected traces, we store the full text of the resulting HTTP response, including both the headers and the body, in a database, keyed based on the hostname used in the request and the URL requested, as well as some meta-information from the headers such as transport and content encoding parameters.

Emulating the Web. On the testbed network, we deploy a very simple web server program to emulate the Web using the data collected above. Upon receiving an HTTP request, it first parses out the hostname, URL, and other parameters, then looks up the corresponding HTTP response text in the database, and finally sends this response back to the client. The content from the database can be distributed across several web servers to provide the ability to handle large traffic loads as well as provide more realistic network characteristics for the traffic.

To give the impression of a much larger network than can be realistically installed on a testbed, we typically deploy this web server on several Linux

machines, each configured with hundreds or even thousands of virtual network interfaces for each physical interface. Each machine can thus respond to HTTP requests sent to any one of thousands of IP addresses. Each instance of the web server application listens on a designated subset of the host's IP addresses and serves content for a designated set of web sites. This flexibility enables us to emulate both very simple sites hosted at a single IP address as well as dynamic, world-wide content distribution networks. We store the mapping from hostnames to IP addresses and Linux hosts in the testbed's central LARIAT database. There, this information is also used to configure the testbed's DNS servers, so that client nodes can resolve hostnames to the proper virtual IP addresses. We also provide artificial Root DNS servers as well as a core BGP routing infrastructure to redistribute all of the routing information for these IP addresses.

Discussion and Limitations. This combination of lightweight application-level replay on the server side with automation of heavyweight GUI applications on the client side allows us to generate very high-fidelity network traffic for many use cases. It requires no parsing or understanding of JavaScript, but many JavaScript-heavy sites can be emulated using this method and appear fully functional from the client's perspective, limited only by the extent of the data collection. One notable example of such a site is Google Maps.

However, the focus on light weight and efficiency in our server-side replay techniques leads to some important limitations of the current implementation. First, because the server is stateless, it cannot do HTTP authorization or any customization of page content based on cookies. Second, because it only looks for exact matches in the URLs, some pages that dynamically generate links may fail to find a matching page when run on the testbed. Pages that dynamically choose IP addresses or hostnames for links may need to be fetched tens or even hundreds of times during the data collection step in order to find all IP addresses or hostnames that should occur in the linked pages' URLs. Otherwise, any client-side JavaScript code that uses random numbers to control its actions (e.g. client-side load balancing) will fail to function given that previously unrequested URLs will not be found in the new closed environment. Finally, while our approach could be almost trivially extended to support the concurrent use of multiple browsers or multiple operating systems, it does not do so currently.

Despite these limitations, the current techniques are sufficient for many kinds of experiments involving network traffic. They are also valuable for tests that focus primarily on host behavior, as they enable a wider range of applications to be run on the host, most notably the web browser. In the next section, we walk through a simple experiment with a host-based security system where use of the browser makes up a significant fraction of the client machine's workload.

4 An Example Experiment

In this section we walk through a simple experiment as an example of the kind of test our system enables a researcher to perform. Although the underlying

LARIAT test range automation tool and the AUSM-based workload generators are capable of scaling to hundreds or even thousands of nodes, for ease of exposition, we will limit ourselves to a much more limited test scenario in this paper. Despite its small scale and relative simplicity, we believe this experiment is still complex enough to illustrate the difficulties in applying the scientific method to problems in computer security.

Specifically, the goal of our example experiment is to measure and quantify the performance penalty incurred by running anti-virus protection on desktop computers. In principle, very nearly the same experiment could be used to measure the performance impact of many other security tools, including other kinds of intrusion detection systems such as content-based filters, or many types of malware like rootkits, adware, spyware, or key loggers. We chose to measure the impact of AV systems because (1) they are ubiquitous on Internet-connected machines, and (2) because although users have long complained that AV negatively impacts system performance, very little hard data has been presented to either refute or support this claim.

Hypothesis. We begin by defining a falsifiable hypothesis. A simple statement such as *"anti-virus makes the system slow"* is not a good hypothesis because slowness is subjective and is therefore not *measurable* without a substantial user study; it depends not only on the performance of the system but also on the perception of the user. Instead, we use a similar hypothesis that we hope will be a good predictor of perceived slowness, namely that "*anti-virus increases the system's resource consumption.*" In related work, others have tested a similar hypothesis, namely that "*anti-virus increases the time required to complete a suite of computational tasks*" [53,54].

4.1 Testbed Setup

We use a very simple experimental testbed comprised of two physical machines. One of these machines is a server-class machine (HOST) which we use to provide the LARIAT infrastructure. HOST is a Dell PowerEdge 2650 with dual Intel Xeon 2.60GHz processors and 2GB of RAM. On HOST, we deploy two virtual servers using VMWare Server. One of these is the LARIAT control server (CONTROL) and the other (INTERNET) provides email, DNS, and world-wide web services (Section 3.2) on the testbed for use by the system under test. The second machine in our setup is a Dell Latitude D610 laptop (SUT, for "system under test") with a 1.7GHz Pentium M processor and 1GB of RAM. We partition the hard disk of the SUT into two pieces. On one partition, we install Ubuntu Linux 9.10. On the other, we install Windows XP with Service Pack 2, Microsoft Office XP, and our client-side workload generation tools, including the login module, the user agent, and the AUSMs for Internet Explorer, Word, Excel, PowerPoint, and Outlook. To enable the collection of performance measurements from the system under test, we install components of SGI's *Performance Co-Pilot* (PCP) software [55] on the Windows partition of the SUT,

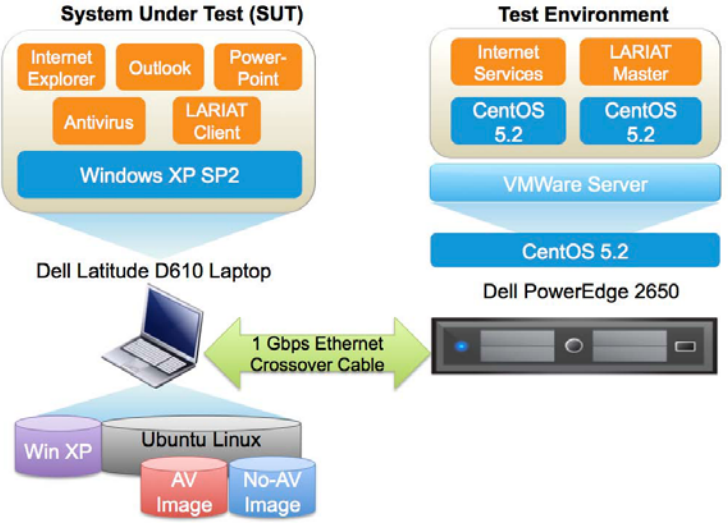


Fig. 3. Test Network Physical Topology

where it can collect performance information, and on HOST, where it can log these measurements for future analysis. We also install the `winexe` remote administration tool on HOST so that we can automatically reboot the laptop when it is in Windows.

Then, from the SUT's Linux installation, we use the Unix tool `dd` to make a byte-level image of the Windows partition and store this as a file in the Linux system. We then re-boot into the Windows system and install an open source anti-virus product, reboot back into the Linux system and make another byte-level copy of the Windows partition with the open source AV product installed. At the completion of this process, we have two Windows disk images on the Linux filesystem: (1) a clean snapshot of the Windows system, with no AV software installed and (2) a snapshot of the clean image with the open source product installed.

4.2 Experimental Methods

In Section 3.1, we explained how we use pseudorandom number generators to drive the AUSMs to deliver repeatable user inputs to the application programs on the testbed. However, more steps are necessary to achieve repeatable results from a system as complex as a modern computer or network. In this section, we discuss the steps we take to improve the repeatability of our very small, simple example experiment. First, we note that we intentionally deploy the SUT on a physical machine instead of using virtualization because of difficulty in obtaining repeatable timing measurements on current VM platforms [56]. Our experimental procedure is outlined in Figure 4; we describe the process in more detail below.

1. **Prepare Systems for Test Run**
   (a) Revert disk images on SUT and INTERNET
   (b) Revert system clocks on SUT and INTERNET
   (c) Reboot SUT laptop into Windows environment
   (d) Seed PRNGs using master experiment seed
   (e) Start PCP performance logging service
2. **Execute Test Run**
   (a) Start AUSM-based client workload generation
   (b) Let workload generation run for 2 hours
   (c) Stop AUSM-based client workload generation
3. **Collect Results**
   (a) Stop PCP performance logging service
   (b) Archive performance logs
   (c) Reboot SUT laptop into Linux environment

Fig. 4. Experimental Procedure

When running a test, we begin with the SUT booted into its Linux environment, HOST powered up, and the INTERNET and CONTROL virtual machines running. We revert the disk images on INTERNET and SUT's Windows partition to clean states, using VMWare's snapshot feature and the Unix `dd` tool, respectively. This is necessary to ensure that SUT's filesystem is unchanged from run to run, and that its view of the Internet, including queued email in the user's inbox, is consistent in each run. Next, we set the system clocks on SUT and INTERNET to a constant value. Setting the clocks before each run is important because many application programs also use pseudorandom number generators, and many of them seed their PRNG with the current value of the clock when they start up. Finally, we set the `GRUB` boot loader on the SUT to load Windows on its next invocation, and we reboot the laptop. While the laptop performs its power-on self test and boots into Windows, we start the PCP performance logger on HOST so that it is ready to begin recording performance metrics from SUT when it comes on line. To maximize repeatability, all of these actions are performed automatically by a set of three shell scripts. One script, the master, runs on HOST and executes the other two scripts to re-set the SUT and the INTERNET VM before launching each run of the experiment.

In each run, the master script sends a command to CONTROL to start the experiment, then sleeps for a specified length of time in order to let the test run. Meanwhile, CONTROL sends the command to SUT, which logs in the virtual user and starts the AUSMs to drive the client-side applications on the testbed. When the master script wakes up, it sends another command to CONTROL, stopping the test and logging the user out. It archives the PCP performance logs and then uses `winexe` to reboot the laptop into its Linux environment in preparation for the next run.

4.3 Experimental Results

Using the above procedure, we select a master experiment seed at random and repeatedly execute a 2-hour test on our testbed with this seed. We run the test

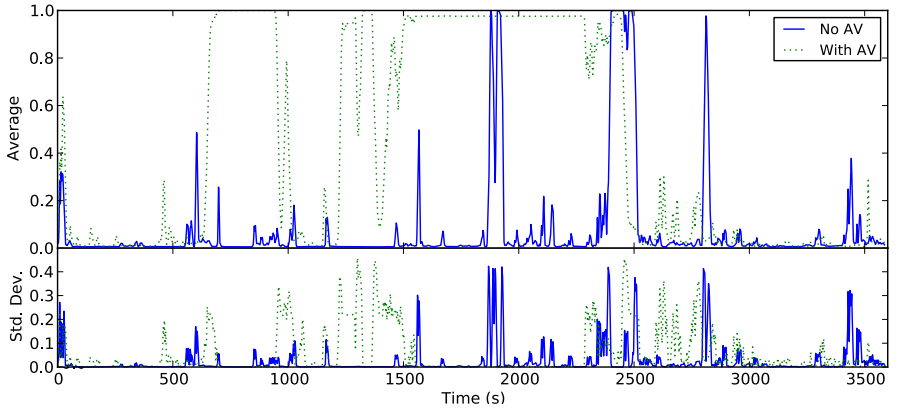


Fig. 5. CPU Utilization

under two distinct scenarios: (1) the baseline, with no anti-virus protection on the SUT and (2) with the open source anti-virus product. These experiments serve two purposes. First, they allow us to test whether there is any measurable difference in system resource consumption between the two scenarios. They also serve to demonstrate the repeatability of results enabled by our techniques.

With the experiment seed that we selected, the user agent launches both Outlook and Word soon after logging in. It spends several minutes reading and writing email and writing Word documents, then launches Internet Explorer and browses to several sites on the emulated web. It keeps these three applications open for the duration of the experiment and frequently switches back and forth between them every few minutes.

For each of the two scenarios, we take all of the runs, and show the average and standard deviation of the CPU utilization (Fig. 5), memory consumption

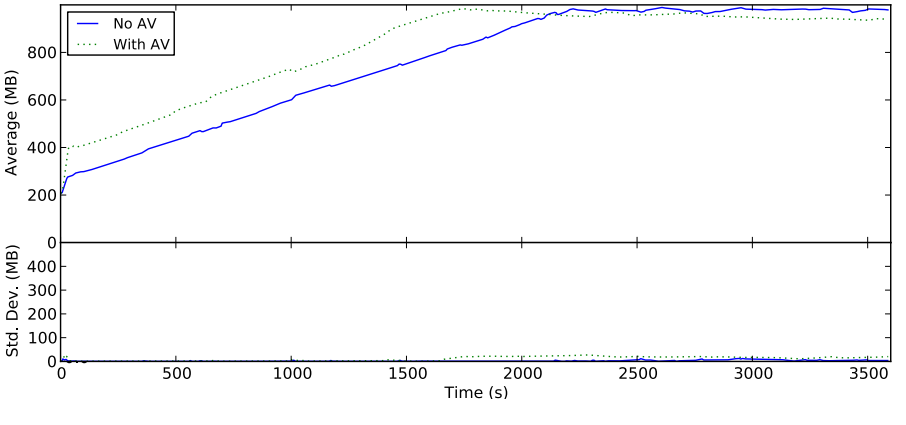


Fig. 6. Memory Consumption

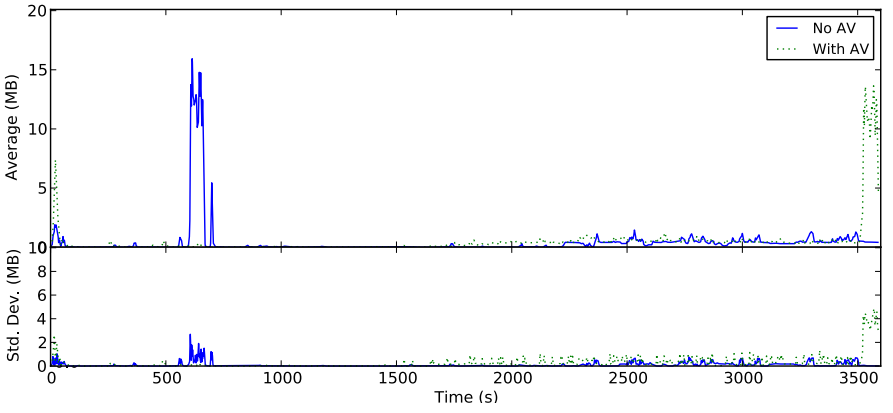


Fig. 7. Disk I/O

(Fig. 6), and disk input/output volume (Fig. 7) for the first hour of the experimental runs. The data was gathered in one second intervals using PCP. In order to make these plots more readable, we have performed a Gaussian smoothing over the data with a 5 second radius. In effect, this smooths out some of the jagged edges in the plot, making them more readable without changing them in any significant way.

Discussion. We see consistent spikes in both CPU load and disk I/O starting near 0 seconds when the user agent launches Outlook and Word, and again near 600 seconds when Internet Explorer is started. In Fig. 5, we see that the open source AV product consistently causes near 100% CPU use for a period of nearly 10 minutes. During this same period, the standard deviation of the CPU utilization is near zero, indicating that this behavior is very repeatable. Throughout Fig. 5 and Fig. 7, spikes in the average measurements are typically accompanied by smaller spikes in the standard deviations. However, we note that during periods of sustained activity, the standard deviation only spikes at the beginning and the end of the plateau in the mean. This pattern occurs in Fig. 5 at 600-1000 seconds, 1200-1400 seconds, and 1500-2500 seconds for the open source product and to a lesser extent from 1800-2000 and 2300-2500 seconds for the baseline case. This leads us to believe that much of the variance in these graphs is due to the inexact timing of our automation scripts.

Figures 5 and 6 show clear evidence of the system with open source anti-virus protection consuming more resources than the same system with no anti-virus, and formal statistical tests confirm our hypothesis with high confidence for these data series. In Fig. 7, overall, the anti-virus system does not appear to cause a statistically significantly increase in disk I/O loads relative to the baseline system. We are interested in whether these same results would hold for commercial anti-virus products, which may be developed with a greater focus on efficiency and performance. In the near future, we may expand the coverage of our test to include one or more commercial tools as well.

5 Conclusions and Future Work

We presented new techniques for driving ubiquitous, commercial-off-the-shelf Windows GUI applications in place of a human user on an isolated testbed network. Using statistical models of user behavior to generate workloads on the testbed hosts, together with dynamic application-level protocol replay techniques to emulate remote services like the World Wide Web, we can generate traffic on the testbed network that resembles real traffic to a very high degree of fidelity.

We demonstrated a small-scale experiment to show how these techniques help to enable configurable, repeatable tests involving client-side security tools, and we highlighted some challenges in achieving repeatable experimental results with such complex systems.

In the future, we plan to improve on our current techniques in a number of ways. First, we will collect data from a larger set of users and develop techniques for validating that the workloads and traffic induced on a testbed faithfully represent the environments they were modeled after. Second, we will develop actuators that require a smaller software footprint on the system under test, to further reduce the risk of test artifacts in experimental results. Finally, we plan to develop more robust techniques for dynamic application-level replay of web sites that make heavy use of JavaScript or other active content generation techniques.

Acknowledgments

The authors extend our sincerest thanks to the members of the MIT-LL Cyber Testing team who implemented much of the software described here and provided much helpful feedback on the experiments and the paper.

References

1. Barford, P., Landweber, L.: Bench-style network research in an Internet Instance Laboratory. ACM SIGCOMM Computer Communication Review 33(3), 21–26 (2003)
2. Peisert, S., Bishop, M.: How to Design Computer Security Experiments. In: Proceedings of the 5th World Conference on Information Security Education (WISE), pp. 141–148 (2007)
3. US Department of Homeland Security: A Roadmap for Cybersecurity Research. Technical report (November 2009), `www.cyber.st.dhs.gov/docs/DHS-Cybersecurity-Roadmap.pdf`
4. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (December 2002)
5. Ricci, R., Duerig, J., Sanaga, P., Gebhardt, D., Hibler, M., Atkinson, K., Zhang, J., Kasera, S., Lepreau, J.: The Flexlab approach to realistic evaluation of networked systems. In: Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation, pp. 201–214 (April 2007)

6. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J., Becker, D.: Scalability and Accuracy in a Large-Scale Network Emulator. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (December 2002)
7. Bavier, A., Feamster, N., Huang, M., Peterson, L., Rexford, J.: VINI veritas: Realistic and controlled network experimentation. In: Proceedings of ACM SIGCOMM (September 2006)
8. Rossey, L.M., Cunningham, R.K., Fried, D.J., Rabek, J.C., Lippmann, R.P., Haines, J.W., Zissman, M.A.: LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed. In: Proceedings of the IEEE Aerospace Conference (2002)
9. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The Ghost in the Browser: Analysis of Web-based Malware. In: Proceedings of the First Workshop on Hot Topics in Understanding Botnets (HotBots 2007) (April 2007)
10. Fossi, M.: Symantec Internet Security Threat Report: Trends for 2008 (April 2009)
11. Deibert, R., Rohozinski, R.: Tracking GhostNet: Investigating a Cyber Espionage Network. Technical Report JR02-2009, Information Warfare Monitor (March 2009)
12. Nagaraja, S., Anderson, R.: The Snooping Dragon: Social-Malware Surveillance of the Tibetan Movement. Technical Report UCAM-CL-TR-746, University of Cambridge Computer Laboratory (March 2009)
13. Provos, N., Mavrommatis, P., Rajab, M., Monrose, F.: All Your iFrames Point to Us. In: Proceedings of the 17th USENIX Security Symposium (July 2008)
14. Pinheiro, E., Weber, W.D., Barroso, L.A.: Failure Trends in a Large Disk Drive Population. In: Proceedings of the 5th USENIX Conference on File and Storage Technologies (February 2007)
15. Lippmann, R.P., Fried, D.J., Graf, I., Haines, J.W., Kendall, K.R., McClung, D., Weber, D., Webster, S.E., Wyschogrod, D., Cunningham, R.K., Zissman, M.A.: Evaluating Intrusion Detection Systems: The 1998 DARPA Off-Line Intrusion Detection Evaluation. In: Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (2000)
16. Lippmann, R., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA Off-line Intrusion Detection Evaluation. Computer Networks 34(4), 279–595 (2000)
17. Yu, T., Fuller, B., Bannick, J., Rossey, L., Cunningham, R.: Integrated Environment Management for Information Operations Testbeds. In: Proceedings of the 2007 Workshop on Visualization for Computer Security (October 2007)
18. Benzel, T., Braden, R., Kim, D., Neuman, C., Joseph, A., Sklower, K., Ostrenga, R., Schwab, S.: Experience with DETER: A Testbed for Security Research. In: Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM) (March 2006)
19. Boothe-Rabek, J.C.: WinNTGen: Creation of a Windows NT 5.0+ network traffic generator. Master's thesis, Massachusetts Institute of Technology (2003)
20. Garg, A., Vidyaraman, S., Upadhyaya, S., Kwiat, K.: USim: A User Behavior Simulation Framework for Training and Testing IDSes in GUI Based Systems. In: ANSS 2006: Proceedings of the 39th Annual Symposium on Simulation, Washington, DC, USA, pp. 196–203. IEEE Computer Society, Los Alamitos (2006)
21. Cui, W., Paxson, V., Weaver, N.C.: GQ: Realizing a System to Catch Worms in a Quarter Million Places. Technical Report TR-06-004, International Computer Science Institute (September 2006)
22. Cui, W., Paxson, V., Weaver, N.C., Katz, R.H.: Protocol-Independent Adaptive Replay of Application Dialog. In: Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006) (February 2006)

23. Small, S., Mason, J., Monrose, F., Provos, N., Stubblefield, A.: To catch a predator: A natural language approach for eliciting malicious payloads. In: Proceedings of the 17th USENIX Security Symposium (August 2008)
24. Wang, K.: Using HoneyClients to Detect New Attacks. In: RECON Conference (June 2005)
25. Wang, Y.M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In: Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006) (February 2006)
26. Sanders, M.: autopy: A simple, cross-platform GUI automation toolkit for Python, `http://github.com/msanders/autopy`
27. Yeh, T., Chang, T.H., Miller, R.C.: Sikuli: Using GUI Screenshots for Search and Automation. In: Proceedings of the 22nd Symposium on User Interface Software and Technology (October 2009)
28. Kleek, M.V., Bernstein, M., Karger, D., Schraefel, M.C.: Getting to Know You Gradually: Personal Lifetime User Modeling (PLUM). Technical report, MIT CSAIL (April 2007)
29. Simpson, C.R., Reddy, D., Riley, G.F.: Empirical Models of TCP and UDP EndUser Network Trafc from NETI@home Data Analysis. In: 20th International Workshop on Principles of Advanced and Distributed Simulation (May 2006)
30. Kurz, C., Hlavacs, H., Kotsis, G.: Workload Generation by Modelling User Behavior in an ISP Subnet. In: Proceedings of the International Symposium on Telecommunications (August 2001)
31. tcpreplay by Aaron Turner, `http://tcpreplay.synfin.net/`
32. Hong, S.S., Wu, S.F.: On Interactive Internet Traffic Replay. In: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (September 2006)
33. Sommers, J., Barford, P.: Self-configuring network traffic generation. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, pp. 68–81 (2004)
34. Cao, J., Cleveland, W.S., Gao, Y., Jeffay, K., Smith, F.D., Weigle, M.C.: Stochastic models for generating synthetic HTTP source traffic. In: INFOCOM (2004)
35. Weigle, M.C., Adurthi, P., Hernández-Campos, F., Jeffay, K., Smith, F.D.: Tmix: a tool for generating realistic TCP application workloads in ns-2. ACM SIGCOMM Computer Communication Review 36(3), 65–76 (2006)
36. Lan, K.C., Heidemann, J.: Rapid model parameterization from traffic measurements. ACM Transactions on Modeling and Computer Simulation (TOMACS) 12(3), 201–229 (2002)
37. Vishwanath, K.V., Vahdat, A.: Realistic and Responsive Network Traffic Generation. In: Proceedings of ACM SIGCOMM (September 2006)
38. Sommers, J., Yegneswaran, V., Barford, P.: Toward Comprehensive Trafc Generation for Online IDS Evaluation. Technical report, University of Wisconsin (2005)
39. Mutz, D., Vigna, G., Kemmerer, R.: An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems. In: Proceedings of the Annual Computer Security Applications Conference (December 2003)
40. Kayacik, H.G., Zincir-Heywood, N.: Generating Representative Traffic for Intrusion Detection System Benchmarking. In: Proceedings of the 3rd Annual Communication Networks and Services Research Conference, pp. 112–117 (May 2005)
41. Sommers, J., Yegneswaran, V., Barford, P.: A framework for malicious workload generation. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, pp. 82–87 (2004)

42. Hunt, G., Brubacher, D.: Detours: Binary Interception of Win32 Functions. In: Third USENIX Windows NT Symposium (July 1999)
43. Klimt, B., Yang, Y.: Introducing the Enron Corpus. In: Proceedings of the First Conference on Email and Anti-Spam (CEAS) (July 2004)
44. Paxson, V., Floyd, S.: Wide Area Traffic: The Failure of Poisson Modeling. IEEE/ACM Transactions on Networking 3(3) (June 1995)
45. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modelling and Computer Simulation 8(1), 3–30 (1998)
46. GINA: MSDN Windows Developer Center, `http://msdn.microsoft.com/en-us/library/aa375457VS.85.aspx`
47. Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., Lepreau, J.: Large-scale Virtualization in the Emulab Network Testbed. In: Proceedings of the 2008 USENIX Annual Technical Conference (June 2008)
48. Google, Inc.: Google search appliance, `http://www.google.com/enterprise/search/gsa.html`
49. osCommerce: Open Source E-Commerce Solutions, `http://www.oscommerce.com/`
50. DMOZ Open Directory Project, `http://www.dmoz.org/`
51. Yahoo! Directory, `http://dir.yahoo.com/`
52. Alexa Top Sites, `http://www.alexa.com/topsites`
53. AV-Comparatives e.V.: Anti-Virus Comparative Performance Test: Impact of Anti-Virus Software on System Performance (December 2009), `http://www.av-comparatives.org/comparativesreviews/performance-tests`
54. Warner, O.: What Really Slows Windows Down (September 2006), `http://www.thepcspy.com/read/what_really_slows_windows_down`
55. Chatterton, D., Gigante, M., Goodwin, M., Kavadias, T., Keronen, S., Knispel, J., McDonell, K., Matveev, M., Milewska, A., Moore, D., Muehlebach, H., Rayner, I., Scott, N., Shimmin, T., Schultz, T., Tuthill, B.: Performance Co-Pilot for IRIX Advanced User's and Administrator's Guide. 2.3 edn. SGI Technical Publications (2002), `http://oss.sgi.com/projects/pcp/index.html`
56. Timekeeping in VMware Virtual Machines, `http://www.vmware.com/pdf/vmware_timekeeping.pdf`

On Challenges in Evaluating Malware Clustering

Peng Li[1], Limin Liu[2], Debin Gao[3], and Michael K. Reiter[1]

[1] Department of Computer Science, University of North Carolina, Chapel Hill, NC, USA
[2] State Key Lab of Information Security, Graduate School of Chinese Academy of Sciences
[3] School of Information Systems, Singapore Management University, Singapore

Abstract. Malware clustering and classification are important tools that enable analysts to prioritize their malware analysis efforts. The recent emergence of fully automated methods for malware clustering and classification that report high accuracy suggests that this problem may largely be solved. In this paper, we report the results of our attempt to confirm our conjecture that the method of selecting ground-truth data in prior evaluations biases their results toward high accuracy. To examine this conjecture, we apply clustering algorithms from a different domain (plagiarism detection), first to the dataset used in a prior work's evaluation and then to a wholly new malware dataset, to see if clustering algorithms developed without attention to subtleties of malware obfuscation are nevertheless successful. While these studies provide conflicting signals as to the correctness of our conjecture, our investigation of possible reasons uncovers, we believe, a cautionary note regarding the *significance* of highly accurate clustering results, as can be impacted by testing on a dataset with a biased cluster-size distribution.

Keywords: malware clustering and classification, plagiarism detection.

1 Introduction

The dramatic growth of the number of malware variants has motivated methods to classify and group them, enabling analysts to focus on the truly new ones. The need for such classification and pruning of the space of all malware variants is underlined by, e.g., the Bagle/Beagle malware, for which roughly 30,000 distinct variants were observed between January 9 and March 6, 2007 [8]. While initial attempts at malware classification were performed manually, in recent years numerous *automated* methods have been developed to perform malware classification (e.g., [11,6,5,16,9,13,15]). Some of these malware classifiers have claimed very good accuracy in classifying malware, leading perhaps to the conclusion that malware classification is more-or-less solved.

In this paper, we show that this may not be the case, and that evaluating automated malware classifiers poses substantial challenges that we believe require renewed attention from the research community. A central challenge is that with the dearth of a well-defined notion of when two malware instances are the "same" or "different", it is difficult to obtain ground truth to which to compare the results of a proposed classifier. Indeed, even manually encoded rules to classify malware seems not to be enough — a previous study [6] found that a majority of six commercial anti-virus scanners concurred on the classification of 14,212 malware instances in only 2,658 cases. However, in the absence of better alternatives for determining ground truth, such instances and

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 238–255, 2010.

their corresponding classifications are increasingly used to evaluate automated methods of malware clustering. For example, a state-of-the-art malware clustering algorithm due to Bayer et al. [6] achieved excellent results using these 2,658 malware instances as ground truth; i.e., the tool obtained results that largely agreed with the clustering of these 2,658 malware instances by the six anti-virus tools.

The starting point of the present paper is the possibility, we conjectured, that one factor contributing to these strong results might be that these 2,658 instances are simply easy to classify, by any of a variety of techniques. We report on our efforts to examine this possibility, first by repeating the clustering of these instances using algorithms from an ostensibly different domain, namely plagiarism detectors that employ dynamic analysis. Intuitively, since plagiarism detectors are developed without attention to the specifics of malware obfuscation, highly accurate clustering results by these tools might suggest that this method of selecting ground-truth data biases the data toward easy-to-classify instances. We describe the results of this analysis, which indicate that plagiarism detectors have nearly the same success in clustering these malware instances, thus providing tentative support for this conjecture.

To more thoroughly examine this possibility, we then attempted to repeat the evaluation methodology of Bayer et al. on a new set of malware instances. By drawing from a database of malware instances, we assembled a set for which four anti-virus tools consistently labeled each member. We detail this study and report on its results that, much to our surprise, find that neither the Bayer et al. technique nor the plagiarism detectors we employed were particularly accurate in clustering these instances. Due to certain caveats of this evaluation that we will discuss, this evaluation is materially different from that for the previous dataset, causing us to be somewhat tentative in the conclusions we draw from it. Nevertheless, these results temper the confidence with which we caution that the selection of ground-truth data based on the concurrence of multiple anti-virus tools biases the data toward easy-to-classify instances.

But this leaves the intriguing question: Why the different results on the two datasets? We complete our paper with an analysis of a factor that, we believe, contributes to (though does not entirely explain) this discrepancy, and that we believe offers a cautionary note for the evaluation of malware clustering results. This factor is the makeup of the ground-truth dataset, in terms of the distribution of the sizes of the malware families it contains. We observe that the original dataset, on which the algorithms we consider perform well, is dominated by two large families, but the second dataset is more evenly distributed among many families. We show that this factor alone biases the measures used in comparing the malware clustering output to the dataset families, specifically precision and recall, in that it increases the likelihood of good precision and recall numbers occurring by chance. As such, the biased cluster-size distribution in the original dataset erodes the *significance* (c.f., [22, Section 8.5.8]) of the high precision and recall reported by Bayer et al. [6]. This observation, we believe, identifies an important factor for which to control when measuring the effectiveness of a malware clustering technique.

While we focus on a single malware classifier for our analysis [6], we do so because very good accuracy has been reported for this algorithm and because the authors of that technique were very helpful in enabling us to compare with their technique. We hasten

to emphasize, moreover, that our comparisons to plagiarism detectors are not intended to suggest that plagiarism detectors are the equal of this technique. For one, we believe the technique of Bayer et al. is far more scalable than any of the plagiarism detectors that we consider here, an important consideration when clustering potentially tens of thousands of malware instances. In addition, the similar accuracy of the technique of Bayer et al. to the plagiarism detectors does not rule out the possibility that the plagiarism detectors are more easily evaded (in the sense of diminishing clustering accuracy); rather, it simply indicates that malware today does not seem to do so. We stress that the issues we identify are not a criticism of the Bayer et al. technique, but rather are issues worth considering for any evaluation of malware clustering and classification.

To summarize, the contributions of this paper are as follows. First, we explore the possibility that existing approaches to obtaining ground-truth data for malware clustering evaluation biases results by isolating those instances that are simple to cluster or classify. In the end, we believe our study is inconclusive on this topic, but that reporting our experiences will nevertheless raise awareness of this possibility and will underline the importance of finding methods to validate the ground-truth data employed in this domain. Second, we highlight the importance of the *significance* of positive clustering results when reporting them. This has implications for the datasets used to evaluate malware clustering algorithms, in that it requires that datasets exhibiting a biased cluster-size distribution not be used as the sole vehicle for evaluating a technique.

2 Classification and Clustering of Malware

To hinder static analysis of binaries, the majority of current malware makes use of obfuscation techniques, notably binary packers. As such, dynamic analysis of such malware is often far more effective than static analysis. Monitoring the behavior of the binary during its execution enables collecting a profile of the operations that the binary performs and offers potentially greater insight into the code itself if obfuscation is removed (e.g., the binary is unpacked) in the course of running it. While this technique has its limitations — e.g., it may be difficult to induce certain behaviors of the malware, some of which may require certain environmental conditions to occur [10,14,19,20] — it nevertheless is more effective than purely static approaches. For this reason, dynamic analysis of malware has received much attention in the research community. Analysis systems such as CWSandbox [25], Anubis [7], BitBlaze [18], Norman [2] and ThreatExpert [1] execute malware samples within an instrumented environment and monitor their behaviors for analysis and development of defense mechanisms.

A common application for dynamic analysis of malware is to group malware instances, so as to more easily identify the emergence of new strains of malware, for example. Such grouping is often performed using machine learning, either by *clustering* (e.g., [6,17,15]) or by *classification* (e.g., [13,5,16,11]), which are unsupervised and supervised techniques, respectively.

Of primary interest in this paper are the methodologies that these works employ to evaluate the results of learning, and specifically the measures of quality for the clustering or classification results. Let M denote a collection of m malware instances to be clustered, or the "test data" in the case of classification. Let $\mathcal{C} = \{C_i\}_{1 \leq i \leq c}$ and

$\mathcal{D} = \{D_i\}_{1 \le i \le d}$ be two partitions of M, and let $f : \{1 \dots c\} \to \{1 \dots d\}$ and $g : \{1 \dots d\} \to \{1 \dots c\}$ be functions. Many prior techniques evaluated their results using two measures:

$$\mathsf{prec}(\mathcal{C}, \mathcal{D}) = \frac{1}{m} \sum_{i=1}^{c} |C_i \cap D_{f(i)}|$$

$$\mathsf{recall}(\mathcal{C}, \mathcal{D}) = \frac{1}{m} \sum_{i=1}^{d} |C_{g(i)} \cap D_i|$$

where $\mathcal{C}$ is the set of clusters resulting from the technique being evaluated and $\mathcal{D}$ is the clustering that represents the "right answer".

More specifically, in the case of classification, C_i is all test instances classified as class i, and D_i is all test instances that are "actually" of class i. As such, in the case of classification, $c = d$ and f and g are the identity functions. As a result $\mathsf{prec}(\mathcal{C}, \mathcal{D}) = \mathsf{recall}(\mathcal{C}, \mathcal{D})$, and this measure is often simply referred to as *accuracy*. This is the measure used by Rieck et al. [16] to evaluate their malware classifier, and Lee et al. [13] similarly uses *error rate*, or one minus the accuracy.

In the clustering case, there is no explicit label to define the cluster in $\mathcal{D}$ that corresponds to a specific cluster in $\mathcal{C}$, and so one approach is to define

$$f(i) = \arg\max_{i'} |C_i \cap D_{i'}|$$

$$g(i) = \arg\max_{i'} |C_{i'} \cap D_i|$$

In this case, f and g will not generally be the identity function (or even bijections), and so precision and recall are different. This approach is used by Rieck et al. [17] and Bayer et al. [6] in evaluating their clustering techniques. In this case, when it is desirable to reduce these two measures into one, a common approach (e.g., [17]) is to use the F-measure:

$$\mathsf{F\text{-}measure}(\mathcal{C}, \mathcal{D}) = \frac{2 \cdot \mathsf{prec}(\mathcal{C}, \mathcal{D}) \cdot \mathsf{recall}(\mathcal{C}, \mathcal{D})}{\mathsf{prec}(\mathcal{C}, \mathcal{D}) + \mathsf{recall}(\mathcal{C}, \mathcal{D})}$$

This background is sufficient to highlight the issues on which we focus in the paper:

Production of $\mathcal{D}$: A central question in the measurement of precision and recall is how the reference clustering $\mathcal{D}$ is determined. A common practice is to use an existing anti-virus tool to label the malware instances M (e.g., [16,13,11]), the presumption being that anti-virus tools embody hand-coded rules to label malware instances and so are a good source of "manually verified" ground truth. Unfortunately, existing evidence suggests otherwise, in that it has been shown that anti-virus engines often disagree on their labeling (and clustering) of malware instances [5]. To compensate for this, another practice has been to restrict attention to malware instances M on which multiple anti-virus tools agree (e.g., [6]). Aside from substantially reducing the number of instances, we conjecture that this practice might contribute to more favorable evaluations of malware classifiers, essentially by limiting evaluations to easy-to-cluster instances. To demonstrate this possibility, in Section 3 we consider malware instances selected in this way

and show that they can be classified by plagiarism detectors (designed without attention to the subtleties of malware obfuscation) with precision and recall comparable to that offered by a state-of-the-art malware clustering tool.

Distribution of cluster sizes in $\mathcal{C}$ and $\mathcal{D}$: In order to maximize both precision and recall (and hence the F-measure), it is necessary for $\mathcal{C}$ and $\mathcal{D}$ to exhibit similar cluster-size distributions; i.e., if one of them is highly biased (i.e., has few, large clusters) and the other is more evenly distributed, then one of precision or recall will suffer. Even when they exhibit similar cluster-size distributions, however, the degree to which that distribution is biased has an effect on the *significance* (e.g., [22, Section 8.5.8]) that one can ascribe to high values of these measures. Informally, the significance of a given precision or recall is related to the probability that this value could have occurred by random chance; the higher the probability, the less the significance. We will explore the effect of cluster-size distribution on significance, and specifically the impact of cluster-size distribution on the sensitivity of the F-measure to perturbations in the distance matrix from which the clustering $\mathcal{C}$ is derived. We will see that all other factors held constant, good precision and recall when the reference clusters in $\mathcal{D}$ are of similar size is more significant than if the cluster sizes are biased. That is, small perturbations in the distance matrix yielding $\mathcal{C}$ tends to decay precision and recall more than if $\mathcal{D}$ and $\mathcal{C}$ are highly biased.

We will demonstrate this phenomenon using the malware clustering results obtained from the state-of-the-art malware clustering tool due to Bayer et al., which obtains very different results on two malware datasets, one with a highly biased clustering and one with a more even clustering. While this is not the only source of variation in the datasets, and so the different results cannot be attributed solely to differences in cluster size distributions, we believe that the cluster size distribution is a factor that must be taken into account when reporting malware clustering results.

3 A Potential Hazard of Anti-virus Voting

As discussed in Section 2, a common practice to produce the ground-truth reference clustering $\mathcal{D}$ for evaluating malware clustering algorithms is to use existing anti-virus tools to label the malware instances and to restrict attention to malware instances M on which multiple anti-virus tools agree. The starting point of our study is one such ground-truth dataset, here denoted BCHKK-data, that was used by Bayer et al. for evaluating their malware clustering technique [6]. Using this dataset, their algorithm, here denoted BCHKK-algo, yielded a very good precision and recall (of 0.984 and 0.930, respectively). BCHKK-data consists of $2,658$ malware instances, which is a subset of $14,212$ malware instances contributed between October 27, 2007 and January 31, 2008 by a number of security organizations and individuals, spanning a wide range of sources (such as web infections, honeypots, botnet monitoring, and other malware analysis services). Bayer et al. ran six different anti-virus programs on these $14,212$ instances, and a subset of $2,658$ instances on which results from the majority of these anti-virus programs agree were chosen to form BCHKK-data for evaluation of their clustering technique BCHKK-algo. Bayer et al. explained that such a subset was chosen because

they are the instances on which ground truth can be obtained (due to agreement by a majority of the anti-virus programs they used).

This seems to be a natural way to pick M for evaluation, as they are the only ones for which the ground-truth clustering (i.e., $\mathcal{D}$) could be obtained with good confidence. However, this also raises the possibility that the instances on which multiple anti-virus tools agree are just the malware instances that are relatively easy to cluster, while the difficult-to-cluster instances are filtered out of M. If this were the case, then this could contribute to the high precision and recall observed for the BCHKK-data dataset, in particular.

Unfortunately, we are unaware of any accepted methodology for testing this possibility directly. So, we instead turn to another class of clustering tools derived without attention to malware clustering, in order to see if they are able to cluster the malware instances in BCHKK-data equally well. Specifically, we apply plagiarism detectors to the BCHKK-data to see if they can obtain good precision and recall.

3.1 Plagiarism Detectors

Plagiarism detection is the process of detecting that portions within a work are not original to the author of that work. One of the most common uses of software plagiarism detection is to detect plagiarism in student submissions in programming classes (e.g., Moss [4], Plaque [24], and YAP [26]). Software plagiarism detection and malware clustering are related to one another in that they both attempt to detect some degree of similarity in software programs among a large number of instances. However, due to the uniqueness of malware samples compared to software programs in general (e.g., in using privileged system resources) and due to the degree of obfuscation typically applied to malware instances, we did not expect plagiarism detectors to produce good results when clustering malware samples.

Here we focus on three plagiarism detectors that monitor dynamic executions of a program. We do not include those applying static analysis techniques as they are obviously not suitable for analyzing (potentially packed) malware instances.

- APISeq: This detector, proposed by Tamada et al. [21], computes the similarity of the sequences of API calls produced by two programs to determine if one is plagiarized from the other. Similarity is measured by using string matching tools such as diff and CCFinder [12].
- SYS3Gram: In this detector, due to Wang et al. [23], short sequences (specifically, triples) of system calls are used as "birthmarks" of programs. Similarity is measured as the Jacaard similarity of the birthmarks of the programs being compared, i.e., as the ratio between the sizes of two sets: (i) the intersection of the birthmarks from the two programs, and (ii) the union of the birthmarks from the two programs.
- API3Gram: We use the same idea as in SYS3Gram and apply it to API calls to obtain this plagiarism detector.

We emphasize that the features on which these algorithms detect plagiarism are distinct from those employed by BCHKK-algo. Generally, the features adopted in BCHKK-algo are the operating system objects accessed by a malware instance, the operations that

were performed on the objects, and data flows between accesses to objects. In contrast, the features utilized by the plagiarism detectors we adopted here are system/API call sequences (without specified argument values).

3.2 Results

We implemented these three plagiarism detectors by following the descriptions in the corresponding papers and then applied the detectors to BCHKK-data (instances used by Bayer et al. [6] on which multiple anti-virus tools agree). More specifically, each detection technique produced a distance matrix; we then used single-linkage hierarchical clustering, as is used by BCHKK-algo, to build a clustering $\mathcal{C}$, stopping the hierarchical clustering at the point that maximizes the p-value of a chi-squared test between the distribution of sizes of the clusters in $\mathcal{C}$ and the cluster-size distribution that BCHKK-algo induced on BCHKK-data.[1] We then evaluated the resulting clustering $\mathcal{C}$ by calculating the precision and recall with respect to a reference clustering $\mathcal{D}$ that is one of

- AV: clustering produced by multiple anti-virus tools, i.e., $\mathcal{D}$ in the evaluation clustering ("ground truth") in Bayer et al.'s paper [6];
- BCHKK-algo: clustering produced by the technique of Bayer et al., i.e., $\mathcal{C}$ in the evaluation in Bayer et al.'s paper [6].

To make a fair comparison, the three plagiarism detectors and BCHKK-algo obtain system information (e.g., API call, system call, and system object information) from the same dynamic traces produced by Anubis [7]. Results of the precision and recall are shown in Table 1.

Table 1. Applying plagiarism detectors and malware clustering on BCHKK-data

$\mathcal{C}$	$\mathcal{D}$	$\mathsf{prec}(\mathcal{C},\mathcal{D})$	$\mathsf{recall}(\mathcal{C},\mathcal{D})$	$\mathsf{F\text{-}measure}(\mathcal{C},\mathcal{D})$
BCHKK-algo	AV	0.984	0.930	0.956
APISeq		0.965	0.922	0.943
API3Gram		0.978	0.927	0.952
SYS3Gram		0.982	0.938	0.960
APISeq	BCHKK-algo	0.988	0.939	0.963
API3Gram		0.989	0.941	0.964
SYS3Gram		0.988	0.938	0.963

One set of experiments, shown where $\mathcal{D}$ is set to the clustering results of BCHKK-algo in Table 1, compares these plagiarism detectors with BCHKK-algo directly. The high (especially) precisions and recalls show that the clusterings produced by these plagiarism detectors are very similar to that produced by BCHKK-algo. A second set of

[1] More specifically, this chi-squared test was performed between the cluster-size distribution of $\mathcal{C}$ and a parameterized distribution that best fit the cluster-size distribution that BCHKK-algo induced on BCHKK-data. The parameterized distribution was Weibull with shape parameter $k = 0.4492$ and scale parameter $\lambda = 5.1084$ (p-value $= 0.8763$).

experiments, shown where $\mathcal{D}$ is set to AV, compares the precisions and recalls of all four techniques to the "ground truth" clustering of BCHKK-data. It is perhaps surprising that SYS3Gram performed as well as it did, since a system-call-based malware clustering algorithm [13] tested by Bayer et al. performed relatively poorly; the difference may arise because the tested clustering algorithm employs system-call arguments, whereas SYS3Gram does not (and so is immune to their manipulation). That issue aside, we believe that the high precisions and recalls reported in Table 1 provide support for the conjecture that the malware instances in the BCHKK-data dataset are likely relatively simple ones to cluster, since plagiarism detectors, which are designed without attention to the specific challenges presented by malware, also perform very well on them.

4 Replicating Our Analysis on a New Dataset

Emboldened by the results in Section 3, we decided to attempt to replicate the analysis of the previous section on a new dataset. Our goal was to see if another analysis would also support the notion that selecting malware instances for which ground-truth evidence is inferred by "voting" by anti-virus tools yields a ground-truth dataset that all the tools we considered (BCHKK-algo and plagiarism detectors alike) could cluster well.

4.1 The New Dataset and BCHKK-algo Clustering

To obtain another dataset, we randomly chose $5,121$ instances from the collection of malware instances from VX heavens [3]. We selected the number of instances to be roughly twice the $2,568$ instances in BCHKK-data. We submitted this set of instances to Bayer et al., who kindly processed these instances using Anubis and then applied BCHKK-algo to the resulting execution traces and returned to us the corresponding distance matrix. This distance matrix covered $4,234$ of the $5,121$ samples; Anubis had presumably failed to produce meaningful execution traces for the remainder.

In order to apply the plagiarism detectors implemented in Section 3 to this data, we needed to obtain the information that each of those techniques requires, specifically the sequences of system calls and API calls for each malware instance. As mentioned in Section 3, we obtained this information for the BCHKK-data dataset via Anubis; more specifically, it was already available in the BCHKK-data data files that those authors provided to us. After submitting this new dataset to the Anubis web interface, however, we realized that this information is not kept in the Anubis output by default. Given that obtaining it would then require additional imposition on the Anubis operators to customize its output and then re-submitting the dataset to obtain analysis results (a lengthy process), we decided to seek out a method of extracting this information locally. For this purpose, we turned to an alternative tool that we could employ locally to generate API call traces from the malware instances, namely CWSandbox [25]. CWSandbox successfully processed (generated non-empty API call traces) for $4,468$ of the $5,121$ samples, including $3,841$ of the $4,234$ for which we had results for the BCHKK-algo algorithm.

We then scanned each of these $3,841$ instances with four anti-virus programs (Activescan 2.0, Nod32 update 4956, Avira 7.10.06.140 and Kaspersky 6.0.2.690). Analyzing the results from these anti-virus programs, we finally obtained $1,114$ malware instances for which the four anti-virus programs reported the same family for each; we denote these $1,114$ as VXH-data in the remainder of this paper. More specifically, each instance is given a label (e.g, Win32.Acidoor.b, BDS/Acidoor.B) when scanned by an anti-virus program. The family name is the generalized label extracted from the instance label based on the portion that is intended to be human-readable (e.g., the labels listed would be in the "Acidoor" family). We defined a reference clustering $\mathcal{D}$ for this dataset so that two instances are in the same cluster $D \in \mathcal{D}$ if and only if all of the four anti-virus programs concurred that these instances are in the same family.[2] Our method for assembling the reference clustering for VXH-data is similar to that used to obtain the reference clustering of BCHKK-data [6], but is more conservative.[3]

We obtained the BCHKK-algo clustering of VXH-data by applying single linkage hierarchical clustering to the subset of the distance matrix provided by Bayer et al. corresponding to these instances. In this clustering step, we used the same parameters as in the original paper [6]. To ensure a fair comparison with other alternatives, we confirmed that this clustering offered the best F-measure value relative to the reference VXH-data clustering based on the anti-virus classifications, in comparison to stopping the hierarchical clustering at selected points sooner or later.

4.2 Validation on BCHKK-Data

As discussed above, we resorted to a new tool, CWSandbox (vs. Anubis), to extract API call sequences for VXH-data. In order to gain confidence that this change would not greatly influence our results, we first performed a validation test, specifically to see whether our plagiarism detectors would perform comparably on the BCHKK-data dataset when processed using CWSandbox. In the validation test, we submitted BCHKK-data to CWSandbox to obtain execution traces for each instance. Out of the $2,658$ instances in BCHKK-data, CWSandbox successfully produced traces for $2,099$ of them. Comparing the API3Gram and APISeq clusterings on these $2,099$ samples, first with reference clustering AV and then with the clustering produced using BCHKK-algo (which, again, uses Anubis) as reference, yields the results in Table 2. Note that due to the elimination of some instances, the reference clusterings have fewer clusters than before (e.g., AV now has 68 families instead of 84 originally). Also note that SYS3Gram results are missing in Table 2 since CWSandbox does not provide system call information. However, high F-measure values for the other comparisons suggest that our plagiarism detectors still work reasonably well using CWSandbox outputs.

[2] The VX heavens labels for malware instances are the same as Kaspersky's, suggesting this is the anti-virus engine they used to label.

[3] The method by which Bayer et al. selected BCHKK-data and produced a reference clustering for it was only sketched in their paper [6], but their clarifications enabled us to perform a comparable data selection and reference clustering process, starting from the $3,841$ instances from VX heavens successfully processed by both CWSandbox and the BCHKK-algo algorithm (based on Anubis). This process retained a superset of the 1,114 instances in VXH-data and produced a clustering of which every cluster of VXH-data is a subset of a unique cluster.

Table 2. Applying plagiarism detectors and malware clustering on BCHKK-data. API3Gram and APISeq are based on CWSandbox traces.

$\mathcal{C}$	$\mathcal{D}$	prec$(\mathcal{C},\mathcal{D})$	recall$(\mathcal{C},\mathcal{D})$	F-measure$(\mathcal{C},\mathcal{D})$
API3Gram	AV	0.948	0.918	0.933
APISeq		0.958	0.934	0.946
API3Gram	BCHKK-algo	0.921	0.931	0.926
APISeq		0.937	0.939	0.938

4.3 Results on VXH-Data

In Section 4.1 we described how we assembled the VXH-data dataset and applied BCHKK-algo and the anti-virus tools to cluster it. We now compare the results of the four clustering techniques run on VXH-data: AV from the anti-virus tools, API3Gram (based on CWSandbox), APISeq (based on CWSandbox) and BCHKK-algo (based on Anubis). Results are shown in Table 3. These results again show that the plagiarism detectors produce comparable clustering results to BCHKK-algo when AV is the reference, offering generally greater precision, worse recall, and a similar F-measure.

Table 3. Applying plagiarism detectors and malware clustering on VXH-data

$\mathcal{C}$	$\mathcal{D}$	prec$(\mathcal{C},\mathcal{D})$	recall$(\mathcal{C},\mathcal{D})$	F-measure$(\mathcal{C},\mathcal{D})$
BCHKK-algo	AV	0.604	0.659	0.630
API3Gram		0.788	0.502	0.613
APISeq		0.704	0.536	0.609
API3Gram	BCHKK-algo	0.790	0.826	0.808
APISeq		0.770	0.798	0.784

Surprisingly, however, these measures indicate that both BCHKK-algo and our plagiarism detectors perform more poorly on VXH-data than they did on BCHKK-data. On the face of it, the results in Table 3 do not support the conjecture of Section 3, i.e., that determining a reference clustering of malware instances based on the concurrence of anti-virus engines might bias the reference clustering toward easy-to-cluster instances. After all, were this the case, we would think that *some* method (if not all methods) would do well when AV is used as the reference clustering. Instead, it may simply be the case that the plagiarism detectors and malware clustering tools leverage features for clustering that are more prevalent in BCHKK-data than in VXH-data. In that case, one might thus conclude that these features are not sufficiently reliable for use across a broad range of malware.

Of course, the results of this section must be taken as a bit more speculative, owing to the different systems (CWSandbox and Anubis) from which the malware traces were gathered before being consumed by the clustering techniques we consider. It is true that there is substantial variability in the length and composition of the API sequences gathered by the different tools, in some cases. For example, Figure 1 shows the CDFs of the

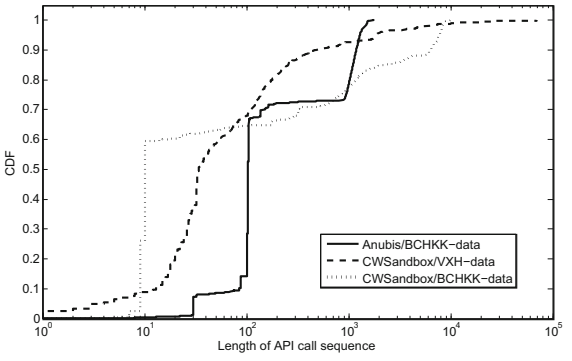


Fig. 1. Lengths of API call sequences extracted from BCHKK-data or VXH-data datasets using CWSandbox or Anubis. Note that x-axis is log-scale.

API call sequence lengths elicited by the different tools. As can be seen in Figure 1, no tool was uniformly better than the other in extracting long API call sequences, though it is apparent that the sequences they induced are very different in length.

Another viewpoint is given in Figure 2, which shows the fraction of malware instances
in each dataset in which certain activities are present. While some noteworthy differences exist, particularly in behaviors related to network activity, it is evident that both tools elicited a range of activities from large portions of the malware datasets. We suspect that some of the differences in frequencies of network activities (particularly "send data" and "receive data") result from the dearth of other malware instances with which to communicate at the time the malware was run in CWSandbox. Again, and despite these differences, our validation tests reported in Table 2 suggest that the sequences induced by each tool are similarly effective in supporting clustering of BCHKK-data.

Activity	Searched Strings	BCHKK-data (Anubis)	BCHKK-data (CWSandbox)	VXH-data (CWSandbox)
create new process	"`CreateProcess`"	100%	87.40%	70.80%
open reg key	"`RegOpenKey`"	100%	95.00%	92.90%
query reg value	"`RegQueryValue`"	100%	94.80%	89.00%
create reg key	"`RegCreateKey`"	98.70%	98.20%	94.20%
set reg value	"`RegSetValue`"	98.30%	97.10%	80.40%
create file	"`CreateFile`"	100%	98.10%	80.60%
send ICMP packet	"`IcmpSendEcho`"	82.10%	82.60%	0.71%
try to connect	"`connect`", "`WSASocket`"	85.10%	89.80%	34.70%
found no host	"`WSAHOST_NOT_FOUND`"	N/A	72.30%	9.06%
send data	"`AFD_SEND`", "`socket_send`"	83.10%	1.50%	14.40%
receive data	"`AFD_RECV`", "`socket_recv`"	83.20%	1.40%	14.90%

Fig. 2. Percentage of malware instances in which listed behavior is observed

The evidence above suggests to us that a different reason for the relatively poor accuracy of BCHKK-algo and our plagiarism detectors on VXH-data is at work. One possible contributing factor is that BCHKK-data samples within the same reference cluster tended to produce API-call sequences of more uniform length than did VXH-data samples in the same reference cluster. For example, the relative standard deviation of the API sequence lengths per cluster in BCHKK-data, averaged over all clusters, is 23.5% and 6.9% for traces produced by Anubis and CWSandbox, respectively, while this number is 130.5% for CWSandbox traces of VXH-data. However, in the following section we focus our attention on another explanation for the poorer clustering performance on VXH-data versus BCHKK-data, and that we believe is more generally instructive.

5 Effects of Cluster-Size Distribution

In seeking to understand the discrepancy between the precision and recall of the BCHKK-algo (and plagiarism-detection) clustering on the BCHKK-data (Section 3) and VXH-data datasets (Section 4), one attribute of these datasets that stood out to us is the distribution of cluster sizes in each. Specifically, the reference clustering for the BCHKK-data is highly biased, in that it contains two large clusters comprising 48.5% and 27% of the malware instances, respectively, and remaining clusters of size at most 6.7%. In contrast, the VXH-data reference dataset is more evenly distributed; the largest cluster in that dataset comprises only 14% of the instances. Figure 3 shows the cluster size distribution of the reference clustering of each dataset; note that the x-axis is log-scale.

The reason that cluster size distribution matters can be seen from an example of clustering 8 points in one of two extreme ways. If when clustering these 8 points, the reference clustering $\mathcal{D}$ comprises two clusters, one of size 7 and one of size 1, then *any* other clustering $\mathcal{C}$ of these 8 points into two clusters of size 7 and 1 is guaranteed to yield $\mathsf{prec}(\mathcal{C}, \mathcal{D})$ and $\mathsf{recall}(\mathcal{C}, \mathcal{D})$ of at least $7/8$. If, on the other hand, $\mathcal{D}$ comprises two clusters of size 4 each, then another clustering $\mathcal{C}$ could yield $\mathsf{prec}(\mathcal{C}, \mathcal{D})$ and $\mathsf{recall}(\mathcal{C}, \mathcal{D})$ as low as $4/8$, and in fact $\binom{4}{2}\binom{4}{2}/\binom{8}{4} = 36/70$ of such clusterings do so. In this sense,

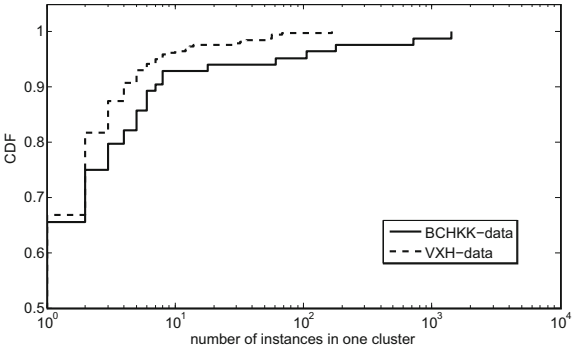


Fig. 3. Reference cluster-size distribution of BCHKK-data and VXH-data. Note that x-axis is log-scale.

it is considerably "harder" to produce a clustering yielding good precision and recall in the latter case, and a good precision and recall in the latter case is thus much more *significant* than in the former.

While providing insight, this combinatorial argument is too simplistic to illustrate the effect that cluster size distribution plays in the BCHKK-algo clustering of the VXH-data and BCHKK-data datasets. A more direct, but still coarse, indication of this effect can be seen by downsampling the large clusters in the BCHKK-data dataset. Specifically, we randomly removed malware instances from the two largest families in the BCHKK-data reference clustering until they were each of size 200. After re-clustering the remaining malware instances using BCHKK-algo with the same parameters, the resulting F-measure averaged over 10 downsampling runs was only 0.815 (versus 0.956 before downsampling).

An alternative and more refined view of the effects of significance to the clustering results of BCHKK-algo for the VXH-data and BCHKK-data datasets can be seen by illustrating the resilience of the clustering results to perturbations in the underlying distance matrix. The heart of the BCHKK-algo clustering technique is the distance measure that it develops, which is tuned to measure the activities of malware. As such, one strategy in examining the potential for bias due to cluster-size distribution is to introduce perturbations into the original BCHKK-algo distance matrices for the VXH-data and BCHKK-data up to some limit, re-cluster the resulting distance matrix into the same cluster-size distribution, and evaluate the rate at which the precision and recall drop. Intuitively, if the precision and recall drop more quickly for the VXH-data than for the BCHKK-data, then this supports the idea that minor errors in the BCHKK-algo distance are more amplified (in terms of the effects on precision and recall) when the clusters are distributed as in the VXH-data than when they are distributed as in the BCHKK-data dataset. By the contrapositive, this will show that a high precision and recall in the VXH-data case is more significant.

In attempting to perform this analysis, however, some difficulties arise.

- The BCHKK-algo distance matrices for the VXH-data and BCHKK-data datasets are different in that the VXH-data matrix results in precision and recall far below that yielded by BCHKK-data. As such, the VXH-data matrix is already "decayed" more from the best possible precision and recall than is that for the BCHKK-data; introducing perturbations in an already decayed distance matrix will do little to demonstrate the sensitivity of a highly accurate distance matrix to perturbations. In order to start from good precision and recall, then, we adopt the *testing* VXH-data matrix and BCHKK-data matrix (i.e., resulting from BCHKK-algo) as the *reference* matrices, i.e., so that we start from precision and recall of 1.0. We then measure the rate of degradation from this ideal as the perturbations are introduced into the distance matrices, compared to these references.
- When re-clustering a perturbed distance matrix, the cluster-size distribution might be altered, in that hierarchical clustering simply might not produce an identical cluster-size distribution as the original from the perturbed distance matrix. For this reason, we fit a parameterized distribution to the reference cluster-size distribution and stop hierarchical clustering at the point that maximizes the p-value of a chi-squared test between the test cluster-size distribution and the fitted reference

distribution. In general, we find that a Weibull distribution with shape parameter $k = 0.7373$ and scale parameter $\lambda = 1.9887$ is a good fit for the reference clustering (i.e., the initial test clustering resulting from BCHKK-algo, as described above) of the VXH-data dataset (p-value of 0.8817), and that the corresponding values for the BCHKK-data are $k = 0.4492$ and $\lambda = 5.1084$ (p-value of 0.8763).

– Given that we have only a distance matrix, a method of perturbing it so that its entries maintain properties of a distance (notably, satisfying the triangle inequality) is necessary. To do this, we map the distance matrix into a d-dimensional space, i.e., creating d-dimensional points to represent the malware instances, separated according to the distances in the matrix. To then perturb the distances, we simply move each point to a random spot in the ball of radius r centered at that point. We can then produce the corresponding distance matrix for these perturbed points, and re-cluster. By increasing r, we then increase the maximum perturbation to the distances.

The results of this analysis are shown in Figure 4. In this figure, the x-axis shows the radius r within which we perturbed each point from its original position in d-dimensional space. The y-axis shows the F-measure that resulted for each radius r, averaged over five runs; standard deviations were negligible. As this figure shows, the cluster-size distributions characteristic of the VXH-data were indeed more sensitive to perturbations in the underlying data than were those characteristic of the BCHKK-data. In addition, in Figure 5 we show the p-values of chi-squared tests comparing the cluster-size distribution of the clustering after perturbation and the fitted (Weibull) reference cluster-size distribution. The fact that these p-values are not significantly decreasing indicates that the cause of degradation in the F-measure was not primarily due to deviation from the intended cluster-size distributions.

We also plot a "Downsized BCHKK-data" line in Figure 4 to control for the discrepancy in the number of malware instances represented in the BCHKK-data and VXH-data datasets. To do this, we randomly removed instances from BCHKK-data (irrespective of the reference clusters in which they fall) until the size of the data set is the same as that of VXH-data, i.e., $1,114$ instances. Using the correspondingly downsized

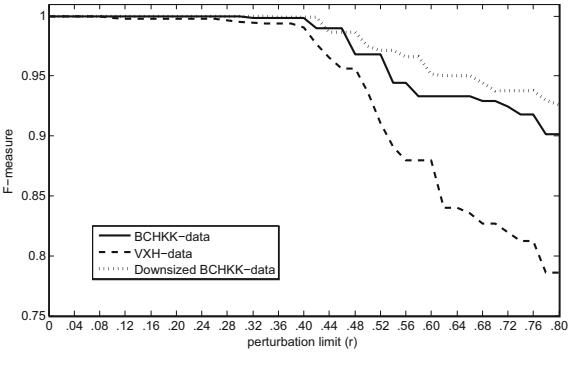


Fig. 4. Tolerance to perturbations

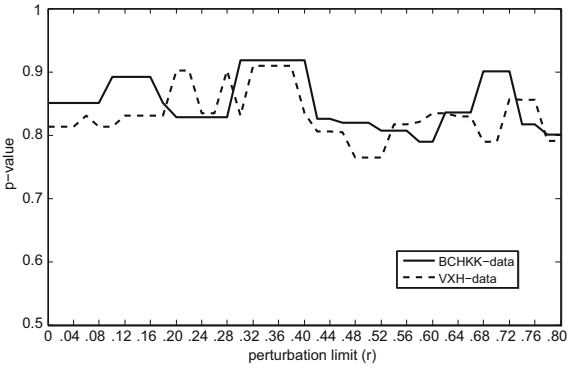


Fig. 5. p-values for perturbation tests

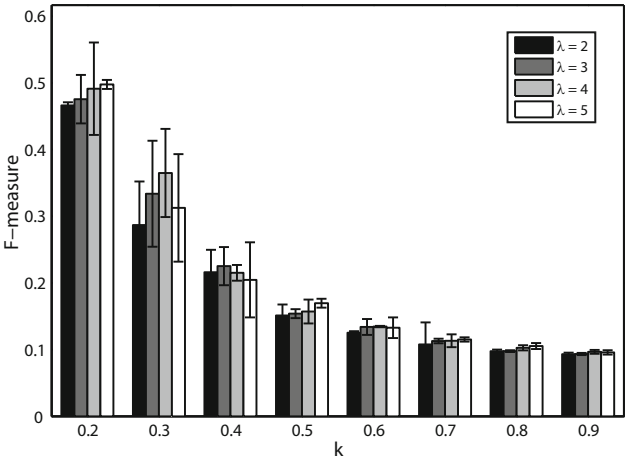


Fig. 6. F-measure of *random* test and reference clusterings with cluster sizes drawn from a Weibull distribution with scale parameter $\lambda \in [2, 5]$ and shape parameter $k \in [0.2, 0.9]$, averaged over 10 trials. Error bars show standard deviation. Note that the best-fit (k, λ) value for the BCHKK-data reference clustering is $(0.4488, 4.8175)$ and for the VXH-data reference clustering is $(0.7803, 2.5151)$.

distance matrix, we applied hierarchical clustering using the same threshold to stop clustering as reported in [6], resulting in a clustering whose cluster-size distribution has corresponding estimated Weibull parameters $k = 0.4307$ and $\lambda = 2.0399$. We took this clustering as the starting point for "Downsized" perturbation test and show the results (averaged over 5 runs) in Figure 4. And as we can see, "Downsized" BCHKK-data is still more immune to perturbation than VXH-data.

To further examine the effects of cluster size distributions on precision and recall, in Figure 6 we plot the average F-measure for reference clusters $\mathcal{D}$ and test clusters $\mathcal{C}$ whose cluster sizes are chosen from a Weibull distribution with the shape parameter

k shown on the x-axis. Once the reference and test cluster sizes are chosen (independently), the reference and test clusters are then populated independently at random (i.e., each point is assigned to a cluster in $\mathcal{D}$ and a cluster in $\mathcal{C}$ independently). As Figure 6 shows, the F-measure that results simply from different values of k provides further insight into the bias that cluster size distribution can introduce.

We do not mean to suggest that the complete discrepancy between the results of the BCHKK-algo clustering technique on the VXH-data and BCHKK-data is due solely to the cluster size distributions underlying the two datasets. However, we do believe that this case and our analysis of it offers sufficient evidence to recommend that evaluation of future clustering techniques be done on datasets with a variety of cluster size distributions. It is also possible that measures of cluster accuracy other than precision and recall better avoid this source of bias. For example, Perdisci et al [15] employed an approach based on the compactness of each cluster and the separation among different clusters, which may be preferable.

6 Conclusion

In this paper we have reported on our investigation of the impact that ground-truth selection might have on the accuracy reported for malware clustering techniques. Our starting point was investigating the possibility that a common method of determining ground truth, namely utilizing the concurrence of multiple anti-virus tools in classifying malware instances, may bias the dataset toward easy-to-cluster instances. Our investigation of this possibility was based on clustering using a different set of tools developed without attention to the subtleties of malware, namely plagiarism detectors. While our application of these tools, first to a dataset used in the evaluation of a state-of-the-art malware clustering technique and second to a whole new malware dataset, arguably leaves our conjecture unresolved, we believe that highlighting this possibility is important to facilitate discussion of this issue in the community.

It has also led us to examine an issue that we believe to be important for future analyses of malware clustering, namely the impact of the ground-truth cluster-size distribution on the significance of results suggesting high accuracy. In particular, we have shown that the highly accurate results reported for a state-of-the-art malware classifier (BCHKK-algo) are tempered by a reduced significance owing to having tested on a dataset with a biased cluster-size distribution. We consequently recommend that future evaluations employ data with a cluster-size distribution that is more even.

We caution the reader from drawing more conclusions from our study than is warranted, however. In particular, despite the similar performance of the BCHKK-algo algorithm and the plagiarism detectors in clustering on the malware datasets we considered, it is not justified to conclude that these algorithms are equally effective for malware clustering. The design of the BCHKK-algo algorithm should make it more difficult to evade, not to mention more scalable. It is evident, however, from our results in Section 3 that either malware today is not designed to exploit differences in the clustering abilities of BCHKK-algo and plagiarism detectors, or else that the ground-truth selection of the test datasets eliminated malware instances that do so.

We recognize that our paper has perhaps introduced more questions than it has definitively answered. Nevertheless, we believe that in addition to the observations above,

multiple opportunities for future research can be drawn from our investigation. In particular, we believe our investigation underlines the importance of further research in malware clustering, specifically in better methods for establishing ground truth, in identifying more reliable features for malware clustering, or in both.

Acknowledgements. We are deeply grateful to Ulrich Bayer, Engin Kirda, Paolo Milani Comparetti, and Christopher Kruegel for helpful assistance, for providing access to the BCHKK-data dataset, for applying BCHKK-algo to our VXH-data dataset, for helpful discussions, and for useful comments on drafts of this paper. This work was supported in part by NSF award CT-0756998 and by DRTech Singapore under grant POD0814140.

References

1. Threatexpert3, http://www.threatexpert.com/
2. Norman sandbox center (2008), http://www.norman.com/security_center/security_tools/en
3. VX Heavens (2010), http://vx.netlux.org/
4. Aiken, A.: Moss: a system for detecting software plagiarism, http://theory.stanford.edu/~aiken/moss/
5. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 178–197. Springer, Heidelberg (2007)
6. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Proceedings of the Network and Distributed System Security Symposium (2009)
7. Bayer, U., Kruegel, C., Kirda, E.: Ttanalyze: A tool for analyzing malware. In: 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference (2006)
8. Commtouch, Inc. Malware outbreak trend report: Bagle/beagle (March 2007), http://www.commtouch.com/documents/Bagle-Worm_MOTR.pdf
9. Gheorghescu, M.: An automated virus classification system. In: Proceedings of the Virus Bulletin Conference, VB (1994)
10. Ha, K.: Keylogger.stawin, http://www.symantec.com/security_response/writeup.jsp?docid=2004-012915-2315-99
11. Hu, X., Chiueh, T., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Proceedings of 16th ACM Conference on Computer and Communications Security (2009)
12. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. IEEE Trans. on Software Engineering, 654–670 (2002)
13. Lee, T., Mody, J.J.: Behavioral classification. In: 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference (2006)
14. McAfee. W97m/opey.c, http://vil.nai.com/vil/content/v_10290.htm
15. Perdisci, R., Lee, W., Feamster, N.: Behavioral clustering of http-based malware and signature generation using malicious network traces. In: USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010 (2010)
16. Rieck, K., Holz, T., Willems, C., Dussel, P., Laskov, P.: Learning and classification of malware behavior. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 108–125. Springer, Heidelberg (2008)

17. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. Technical Report 18-2009, Berlin Institute of Technology (2009)
18. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security (December 2008)
19. Symantec. Spyware.e2give, http://www.symantec.com/security_response/writeup.jsp?docid=2004-102614-1006-99
20. Symantec. Xeram.1664, http://www.symantec.com/security_response/writeup.jsp?docid=2000-121913-2839-99
21. Tamada, H., Okamoto, K., Nakamura, M., Monden, A., Matsumoto, K.: Dynamic software birthmarks to detect the theft of windows applications. In: International Symposium on Future Software Technology (2004)
22. Tan, P., Steinbach, M., Kumar, V.: Introduction to Data Mining. Addison-Wesley, Reading (2006)
23. Wang, X., Jhi, Y., Zhu, S., Liu, P.: Detecting software theft via system call based birthmarks. In: Proceedings of 25th Annual Computer Security Applications Conference (2009)
24. Whale, G.: Identification of program similarity in large populations. Computer Journal, Special Issue on Procedural Programming, 140–146 (1990)
25. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. In: Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P 2007), pp. 32–39 (2007)
26. Wise, M.J.: Detection of similarities in student programs: Yaping may be preferable to plagueing. In: Proceedings of the 23rd SIGCSE Technical Symposium (1992)

Why Did My Detector Do *That*?!
Predicting Keystroke-Dynamics Error Rates

Kevin Killourhy and Roy Maxion

Dependable Systems Laboratory
Computer Science Department
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA 15213
{ksk,maxion}@cs.cmu.edu

Abstract. A major challenge in anomaly-detection studies lies in identifying the myriad factors that influence error rates. In keystroke dynamics, where detectors distinguish the typing rhythms of genuine users and impostors, influential factors may include the algorithm itself, amount of training, choice of features, use of updating, impostor practice, and typist-to-typist variation.

In this work, we consider two problems. (1) Which of these factors influence keystroke-dynamics error rates and how? (2) What methodology should we use to establish the effects of multiple factors on detector error rates? Our approach is simple: experimentation using a benchmark data set, statistical analysis using linear mixed-effects models, and validation of the model's predictions using new data.

The algorithm, amount of training, and use of updating were strongly influential while, contrary to intuition, impostor practice and feature set had minor effect. Some typists were substantially easier to distinguish than others. The validation was successful, giving unprecedented confidence in these results, and establishing the methodology as a powerful tool for future anomaly-detection studies.

Keywords: anomaly detection; keystroke dynamics; experimental methodology.

1 Introduction

Anomaly detectors have great potential for increasing computer security (e.g., detecting novel attacks and insider-type behavior [6]). Unfortunately, the error rates of detection algorithms are sensitive to many factors including changes in environmental conditions, detector configuration, and the adversary's behavior. With so many factors that *might* affect a detector's error rates, how do we find those that do? For anomaly detectors to become a dependable computer-security technology, we must be able to explain what factors influence their error rates and how.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 256–276, 2010.

Consider keystroke dynamics: an application of anomaly detection in which the normal typing rhythms of a genuine user are distinguished from those of an impostor. We might discover an impostor, even though he has compromised the password of a genuine user, because he does not type it with the same rhythm. Like all applications of anomaly detection, various factors might influence the error rates of keystroke-dynamics detectors. We have identified six from the literature:

1. **Algorithm:** The anomaly-detection algorithm itself is an obvious factor. Different algorithms will have different error rates. However, it can be difficult to predict error rates from the algorithm alone. In earlier work, we benchmarked 14 existing algorithms on a single data set [12]. The error rates for each algorithm were different from those reported when the algorithm was first proposed; other factors must influence the error rates.
2. **Training amount:** Some researchers have trained their detectors with as few as 5 repetitions of a password, while others have used over 200. Researchers have found that increasing the number of training repetitions even from 5 to 10 can reduce error [10].
3. **Feature set:** A variety of timing features, including hold times, keydown-keydown times, and keyup-keydown times have been tried. Different researchers use different combinations of these features in their evaluation. One study found that every combination had a different error rate [1].
4. **Updating:** Most research has focused on detectors that build a genuine user's typing profile during a training phase, after which the profile is fixed. Recent research suggests that regularly updating the profile may reduce error because the profile evolves with changing typing behavior [1,11].
5. **Impostor practice:** An impostor is an intelligent adversary and will presumably try to evade detection. Since the typing rhythms of a practiced password may be more similar to the genuine user's rhythms, some researchers have given some of their impostor subjects the opportunity to practice. Preliminary results suggest that impostor practice may raise miss rates [1,13].
6. **Typist-to-typist variation:** Some genuine users may be easy to discriminate from impostors while others may be more difficult. When researchers report per-subject error rates, the results do suggest that a detector's error is higher for some subjects than others, but typist-to-typist variation has never been explicitly quantified [5].

Any of these six factors might explain different keystroke-dynamics error rates.

However, earlier work on the effects of these factors is inconclusive. Usually, only one factor at a time is tested, ignoring the possibility of interactions (e.g., that increased training affects different detectors differently). Evaluation results are almost always presented with no statistical analysis. For instance, a detector's empirically measured false-alarm rate using one set of features may be 1.45% while it is 1.63% with another set of features [1]. Without further analysis (e.g., a test of statistical significance), we should not conclude that the first feature set is better than the second, yet such analysis is rarely conducted.

In keystroke dynamics, as in other applications of anomaly detection, listing a multitude of factors that might explain different error rates is easy. The challenge is establishing which factors actually do have an effect.

2 Problem and Approach

In this work, two problems concern us. First, what influence do each of the factors listed above—algorithm, training amount, feature set, updating, impostor practice, and typist-to-typist variation—have on keystroke-dynamics error rates? Second, what methodology should we use to establish the effects of these various factors?

We propose a methodology and demonstrate that it can identify and measure the effects of these six factors. The details of the methodology, which are described in the next three sections, can be summarized as follows:

1. **Experiment:** We design and conduct an experiment in which anomaly detectors are repeatedly evaluated on a benchmark data set. Between evaluations, the six factors of interest are systematically varied, and the effect on the evaluation results (i.e., the error rates of the detectors) is observed. (Section 3)
2. **Statistical analysis:** The experimental results are incorporated into a statistical model that describes the six factors' influence. In particular, we use a *linear mixed-effects model* to estimate the effect of each factor along with any interactions between the factors. Roughly, the mixed-effects model allows us to express both the effects of some factors we can control, such as the algorithm, and also the effects of some factors we cannot, such as the typist. (Section 4)
3. **Validation:** We collect a new data set, comprised of 15 new typists, and we validate the statistical model. We demonstrate that the model predicts evaluation results using the new data, giving us high confidence in its predictive power. (Section 5)

With such a model, we can predict what different environmental changes, reconfigurations, or adversarial behavior will do to a detector. We can make better choices when designing and conducting future evaluations, and practitioners can make more informed decisions when selecting and configuring detectors.

Fundamentally, the proposed methodology—experimentation, statistical analysis, and validation—enumerates several steps of the classical scientific method. Others have advocated that computer-security research would benefit from a stricter application of this method [15]. The current work explores the specific benefit for anomaly-detection research.

3 Experiment

The experiment is designed so that we can observe the effects of each of the six factors on detector error rates. In this section, we lay out the experimental method, and we present the empirical results.

3.1 Experimental method

The method itself is comprised of three steps: (1) obtain a benchmark data set, (2) select values of interest for each of the six factors, and (3) repeatedly run an evaluation, while systematically varying the factors among the selected values.

Data. For our evaluation data, we used an extant data set wherein 51 subjects typed a 10-character password (`.tie5Roanl`). Each subject typed 400 repetitions of the password over 8 sessions of 50 repetitions each (spread over different days). For each repetition of the password, 31 timing features were extracted: 11 hold times (including the hold for the `Return` key at the end of the password), 10 keydown-keydown times (one for each digram), and 10 keyup-keydown times (also for each digram). As a result, each password has been converted to a 31-dimensional password-timing vector. The data are a sensible benchmark since they are publicly available and the collection methods have been laid out in detail [12].

Selecting Factor Values. The six factors of interest in this study—algorithm, training amount, feature set, updating, impostor practice, and typist-to-typist variation—can take many different values (e.g., amount of training can range from 1 repetition to over 200). For this study, we need to choose a subset of values to test.

1. **Algorithms:** We selected three detectors for our current evaluation. The Manhattan (scaled) detector, first described by Araújo et al. [1], calculates the "city block" distance between a test vector and the mean of the training vectors. The Outlier-count (z-score) detector, proposed by Haider et al. [8], counts the number of features in the test vector which deviate significantly from the mean of the training vectors. The Nearest Neighbor (Mahalanobis) detector, described by Cho et al. [5], finds the distance between the test vector and the nearest vector in the training data (using a measure called the Mahalanobis distance). We focus on these three for pragmatic reasons. All three were top performers in an earlier evaluation [12]; their error rates were indistinguishable according to a statistical test. Finding factors that differentiate these detectors will have practical significance, establishing when one outperforms the others.
2. **Amount of training:** We train with 5 repetitions, 50 repetitions, 100 repetitions, and 200 repetitions. Prior researchers have trained anomaly detectors with varying amounts of data, spanning the range of 5–200 repetitions. Our values were chosen to broadly map the contours of this range.
3. **Feature sets:** We test with three different sets of feature: (1) the full set of 31 hold, keydown-keydown, and keyup-keydown times for all keys including the `Return` key; (2) the set of 19 hold and keydown-keydown times for all keys except the `Return` key; (3) the distinct set of 19 hold and keyup-keydown times for all keys except the `Return` key. Prior work has shown that combining hold times with either keydown-keydown times or keyup-keydown times improves accuracy. The three feature sets we test should remove remaining ambiguity about whether the particular combination matters, and whether the `Return`-key timing features should be included.

4. **Updating:** We test with and without updating. Specifically, we compare detectors given a fixed set of training data to those which are retrained after every few repetitions. We call these two modes of updating *None* and *Sliding Window*. Two levels are all that are necessary to establish whether updating has an effect.
5. **Impostor practice:** We test using two levels of impostor practice: *None* and *Very High*. With no practice, impostor data are comprised of the first five password-timing vectors from the impostors. With very high practice, the last five vectors are used, by which point the impostor has had 395 practice repetitions. To explore whether practice has a detrimental effect, only these two extremes are necessary.
6. **Typist-to-typist variation:** By designating each subject as the genuine user in separate evaluation runs, we can observe how much variation in detector performance arises because some subjects are easier to distinguish than others. Since there are 51 subjects in the benchmark data set, we have 51 instances with which to observe typist-to-typist variation.

These selected values enable us to identify which factors are influential and to quantify their effect.

Evaluation procedure. Having chosen values for these six factors, we need an evaluation procedure that can be run for all the different combinations. The designed procedure has seven inputs: the data set (D); the algorithm (A); the number of training repetitions (T); the feature set (F); the updating strategy (U); the level of impostor practice (I); and the genuine-user subject (S).

For clarity, we first describe the evaluation procedure for the no-updating case (i.e., U is set to None):

1. Unnecessary features are removed from the data set. Based on the setting of F, keydown-keydown, keyup-keydown, and/or `Return`-key features are dropped.
2. The detector is trained on the training data for the genuine user. Specifically, repetitions 1 through T for subject S are extracted and used to train the detection algorithm A.
3. Anomaly scores are calculated for the genuine-user test data. Specifically, repetitions $(T + 1)$ through $(T + 200)$ for subject S are extracted (i.e., the next 200 repetitions). The trained detector processes each repetition and calculates an anomaly score. These 200 anomaly scores are designated *genuine-user* scores.
4. Anomaly scores are calculated for the impostor test data. If I is set to None (unpracticed impostors), repetitions 1 through 5 are extracted from every impostor subject (i.e. all those in the data set except S). If I is set to Very High (practiced impostors), repetitions 396 through 400 are extracted instead. The trained detector processes each repetition and calculates an anomaly score. If there are 50 impostor subjects, this step produces 250 (50×5) anomaly scores. These scores are designated *impostor* scores.

5. The genuine-user and impostor scores are used to generate an ROC curve for the detector [19]. From the ROC curve, the equal-error rate is calculated (i.e., the false-alarm and/or miss rate when the detector has been tuned so that both are equal). It is a common overall measure of detector performance in keystroke-dynamics research [14].

The evaluation procedure for the sliding-window-updating case is more complicated (i.e., when U is set to Sliding Window). The intuition is that we slide a window of size T over the genuine user's typing data (advancing the window in increments of five repetitions for computational efficiency). For each window, the detector is trained on the repetitions in that window and then tested using the next five repetitions. We increment the window and repeat. In total, since there are 200 repetitions of genuine-user test data (see Step 3 above), we iterate through 40 such cycles of training and testing (200/5). In our actual evaluation, each of these 40 training-testing cycles is handled in parallel using 40 separate copies of the detector. Each copy is put through its own version of steps 2, 3, and 4 of the evaluation procedure:

2′. Forty different sets of training data are used to train 40 different copies of the detection algorithm A. The first set of training data is comprised of repetitions 1 through T for subject S; the second set is comprised of repetitions 6 through $(T + 5)$; the 40th set is comprised of repetitions 196 through $(T + 195)$. For each of the 40 sets, a separate copy of the detector is trained.

3′. Anomaly scores are calculated for each of 40 different sets of genuine-user test data. Each set corresponds to the genuine-user test data for one of the trained detectors. In particular, the first set includes repetitions $(T + 1)$ through $(T + 5)$; the second set includes $(T + 6)$ through $(T + 10)$; the 40th set includes $(T + 196)$ through $(T + 200)$. The first trained detector scores the 5 repetitions in the first set; the second trained detector scores the 5 repetitions in the second set, and so on. The scores from every detector are pooled together. Since each set contains 5 repetitions and there are 40 sets, there are 200 (5×40) *genuine-user* scores in total.

4′. Anomaly scores are calculated for the impostor test data by every one of the 40 different trained detectors. Specifically, the impostor test data are selected according to the setting of I (i.e., either the first 5 or the last 5 repetitions from every subject except S). Each of the 40 trained detectors scores the repetitions in the impostor test data. If there are 50 impostor subjects and 5 repetitions per subject, this step produces 10,000 $(50 \times 5 \times 40)$ anomaly scores. All of these scores are pooled into a single set of *impostor* scores.

As in the case of no updating, the genuine-user and impostor scores are used to generate a single ROC curve and calculate a single equal-error rate for the sliding-window evaluation.

A few decisions in the design of the sliding-window procedure are worth highlighting. In Step 2′, we make the simplifying assumption that the detector will only retrain on the genuine user's data (i.e., impostor poisoning of the training

is not considered). In Step 4′, we score each repetition of impostor test data multiple times, once with each trained detector. An impostor's anomaly scores will change whenever the detector is retrained. By scoring at each detector window and pooling, we effectively aggregate over these variations and find the average.

We ran this evaluation procedure 7,344 times ($3 \times 4 \times 3 \times 2 \times 2 \times 51$), once for each combination of algorithm, amount of training, feature set, updating, impostor practice, and subject in the data set. We recorded the equal-error rate from each evaluation. By looking at all the combinations of the six factors, we will be able to find interactions between factors, not just the effect of each factor individually.

3.2 Results

To visually explore the 7,344 equal-error rates that comprise the raw results of our experiment, we calculated the average equal-error rate across all subjects for each combination of the other five factors. These averages are presented across the 12 panels in Figure 1. Each panel contains three curves, depicting how the error rates of each of the three detectors changes with increased amounts of training. The strips above each panel explain what combination of factors produced the results in the panel. Updating is either None or Sliding Window; Feature Set is one of the combinations of hold times (H), keydown-keydown times (DD), keyup-keydown times (UD), and `Return`-key features (Ret); Impostor Practice is either None or Very High.

Looking within any panel, we see that the error rates for all three detectors decrease as training increases. In particular, the Nearest Neighbor (Mahalanobis) error is much higher with only 5 repetitions, but improves and is competitive with the others with 100–200 repetitions. With few training repetitions, a practitioner might want to use either the Manhattan (scaled) or Outlier-score (z-count) detector.

If we look beyond the individual panels to the four quadrants, the three panels in a quadrant correspond to the use of the three different feature sets. The curves in the three panels in each quadrant look nearly identical. It would appear that, so long as hold times and one of keydown-keydown or keyup-keydown times are used, the particular combination does not matter.

The six panels on the left correspond to unpracticed-impostor error rates, and the six on the right correspond to very-practiced-impostor error rates. The curves in the right-hand panels are slightly higher. Consequently, impostor practice may represent a minor threat to the accuracy of keystroke-dynamics detectors.

Finally, the six panels on the top correspond to non-updating detectors and the six on the bottom correspond to a sliding-window updating strategy. In particular, at 5 and 50 repetitions, the error-rate curves are much lower in the lower panels. An updating strategy seems to improve performance, especially if operating with only a few training repetitions.

Overall, the lowest average equal-error rate was 7.1%, observed for the Manhattan (scaled) detector with 100 repetitions of training, hold and keydown-keydown features, sliding-window updating, and unpracticed impostors. Among

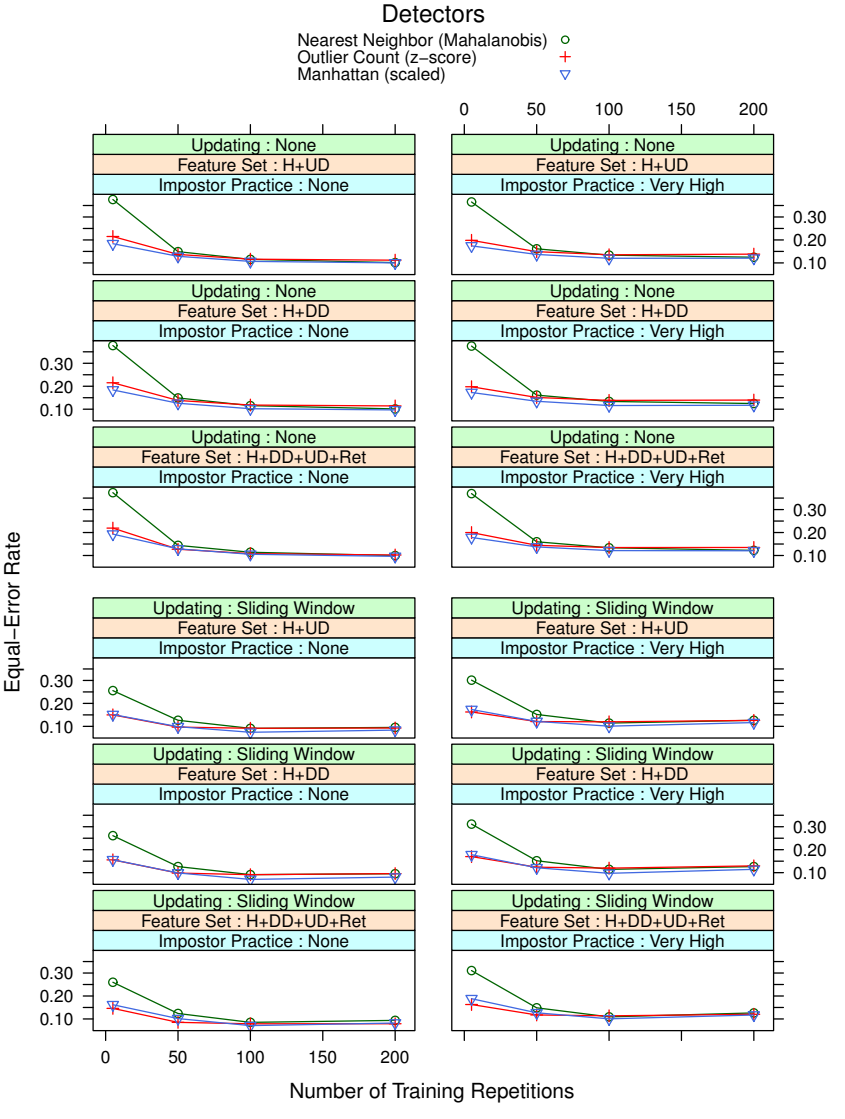


Fig. 1. The average equal-error rates for each detector in the experiment as a function of training amount, feature set, updating strategy, and impostor practice. Each curve shows the effect of training on one of the three detectors. Each panel displays the results for one combination of updating (None/Sliding Window), feature set (H:hold times, DD:keydown-keydown times, UD:keyup-keydown times, Ret:`Return`-key times), and impostor practice (None/Very High). Comparisons can be made across panels because the scales are the same. For instance, the error-rate curves in the upper six panels (no updating) are higher than the curves in the lower six panels (sliding-window updating). This comparison suggests updating reduces error rates.

the very-practiced impostor results, the lowest average equal-error rate was 9.7%, observed for the same combination of algorithm, training amount, feature set, and updating. The empirical results would seem to recommend this combination of detector, training amount, and feature set, but we would withhold a recommendation without further statistical analysis.

4 Statistical Analysis

The empirical results and visualization in Section 3 provide some insight into what factors might be important, but to make predictions about an anomaly detector's future performance we need a statistical model. In this section, we describe the statistical analysis that we performed, and we present the model that it produced.

4.1 Procedure

The analysis is described in stages. First, we explain what a linear mixed-effects model is. Then, we describe how we estimate the model parameters. Finally, we lay out the procedure for selecting a particular linear mixed-effects model.

Linear mixed-effects models. In statistical language, we intend to model the effect of six factors—algorithm, training amount, feature set, updating strategy, impostor practice, and typist (or subject)—on a response variable: the detector's equal-error rate. *Fixed* and *random* are terms used by statisticians to describe two different kinds of effect. When a model has both fixed and random effects, it is called a *mixed-effects* model.

The difference between fixed and random effects is sometimes subtle, but the following rule of thumb is typically applied. If we care about the effect of each value of a factor, the factor is a fixed effect. If we only care about the variation among the values of a factor, the factor is a random effect.

For instance, we treat the algorithm as a fixed effect and the subject (or typist) as a random effect. Practitioners want to know which algorithm's equal-error rate is lower: Manhattan (scaled) or Nearest Neighbor (Mahalanobis). We care about the effect of each value of the factor, so algorithm is a fixed effect. In contrast, practitioners do not want to know which subject's equal-error rate is lower: Subject 1 or Subject 2. Neither subject will be a typist on their system. What we care about is how much variation there is between typists, and so the subject is a random effect.

The following example of a mixed-effects model for keystroke-dynamics data may further elucidate the difference:

$$\begin{aligned} Y &= \mu + A_h + T_i + F_j + U_k + I_l + S + \epsilon \\ S &\sim N(0, \sigma_s^2) \\ \epsilon &\sim N(0, \sigma_\epsilon^2) \end{aligned} \tag{1}$$

The notation in model equation (1) may seem daunting at first. On the first line of the model, Y is the response (i.e., the equal-error rate); μ is a baseline

equal-error rate; A_h, T_i, F_j, U_k, and I_l are the fixed effects of the algorithm, training amount, feature set, updating strategy, and impostor practice, respectively; S is the random effect of the typist (or subject); and, ϵ is the noise term. On the second line, the distribution of the random effect (S) is assumed to be Normal with zero mean and an unknown variance, denoted σ_s^2. On the third line, the distribution of the noise (ϵ) is Normally distributed with zero mean and a different unknown variance, denoted σ_ϵ^2.

The first term in the model equation (μ) denotes the average equal-error rate for one particular combination of fixed-effect factor values, called the baseline values. For instance, the baseline values might be the Manhattan (scaled) detector, 5 training repetitions, hold and keyup-keydown times, no updating, and unpracticed impostors. The average equal-error rate for that combination is μ (e.g., $\mu = 17.6\%$).

For each of the five fixed effects, there is a separate term in the model equation (A_h, T_i, F_j, U_k, I_l). These terms denote the effect on the equal-error rate of departing from the baseline values. For instance, A_h is a placeholder for either of two departures from the Manhattan (scaled) baseline algorithm : A_1 denotes the change to Outlier-count (z-score) and A_2 denotes the change to Nearest Neighbor (Mahalanobis) detector. If the detector in the baseline combination were replaced with the Outlier-count (z-score) detector, the equal-error rate would be calculated as $\mu + A_1$ (e.g., $17.6\% + 2.7 = 20.3\%$).

For the random effect, there is both a term in the model equation (S) and a distributional assumption ($S \sim N(0, \sigma_s^2)$). Like the fixed-effects terms, S represents the effect of a departure. Specifically, it introduces a per-subject effect that is negative ($S < 0$) for easy-to-discriminate subjects and positive ($S > 0$) for hard-to-discriminate subjects. Unlike the fixed-effects term, S is a random variable centered at zero. Its variance (σ_s^2) expresses a measure of the typist-to-typist variation in the model.

The final term in the model equation (ϵ) is the noise term representing the unknown influences of additional factors on the equal-error rate. Like the random effect (S), ϵ is a Normally distributed random variable. Its variance, σ_ϵ^2 expresses a measure of the residual uncertainty in our equal-error predictions.

Parameter estimation. When fitting a linear mixed-effects model, the unknown parameters (e.g., μ, A_h, T_i, F_j, U_k, I_l, σ_s^2, and σ_ϵ^2) are estimated from the data. There are a variety of accepted parameter-estimation methods; a popular one is the method of maximum-likelihood. From any estimate of the parameters, it is possible to derive a probability density function. Among all estimates, the maximum-likelihood estimates are those which give the greatest probability density to the observed data [18].

However, the maximum-likelihood methods have been shown to produce biased estimates of the variance parameters (e.g., σ_s^2). The favored method is a slight elaboration called REML estimation (for restricted or residual maximum likelihood) which corrects for the bias in maximum-likelihood estimation [16,18]. We adopt the REML estimates.

Model selection. In the discussion so far, we have explained how to interpret model equation (1) and how to do parameter estimation given such an equation. We have not explained how to select that model equation in the first place. For instance, consider the following alternative:

$$\begin{aligned} Y &= \mu + A_h + T_i + AT_{hi} + S + \epsilon \\ S &\sim N(0, \sigma_s^2) \\ \epsilon &\sim N(0, \sigma_\epsilon^2) \end{aligned} \tag{2}$$

In model equation (2), the terms corresponding to feature set (F_j), updating (U_k), and impostor practice (I_l) do not appear, so they are assumed to have no effect. An interaction term between algorithm and training (AT_{hi}) appears, so the effect of training is assumed to depend on the algorithm.

The interaction term denotes the effect on the equal-error rate of a departure from the baseline values in *both* algorithm *and* training amount. Without an interaction term, the effect would be additive ($A_h + T_i$). With an interaction term, the additive effect can be adjusted ($A_h + T_i + AT_{hi}$), increased or decreased as fits the data. Interaction effects can be estimated with REML estimation just like the other parameters.

Looking at the relationship between the algorithm and training in Figure 1, we would expect to see a model with an AT_{hi} interaction. The Nearest Neighbor (Mahalanobis) has a much steeper slope from 5–50 training repetitions than the other two detectors. If the effects of the algorithm and training were additive (i.e., no interaction effect), the slopes of the three curves would be parallel. Consequently, model equation (2) might describe our data better.

Model equations (1) and (2) are but two members of a whole family of possible models that we might use to describe our data. We need a method to search through this family of models and find the one that is most appropriate. Specifically, we need a way to compare two models and decide which one better explains the data.

Of the various model-comparison strategies, one often employed is Schwartz's Bayesian Information Criterion (BIC). The details are beyond the scope of this paper, but in brief a model's BIC score is a summary combining both how well the model fits the data (i.e., the likelihood of the model) and also the number of parameters used to obtain the fit (i.e., the number of terms in the model). Having more parameters leads to a better fitting model, and having fewer parameters leads to a simpler model. BIC captures the trade-off between fit and simplicity. When comparing two models, we calculate and compare their BIC scores. Of the two, we adopt the one with the lower score [9].

Let us note one procedural issue when performing BIC-based model selection using mixed-effects models. REML estimation is incompatible with this heuristic, and so when comparing two models using BIC, the maximum-likelihood estimates are used. Once a model is selected, the parameters are re-estimated using REML. Intuitively, we use the maximum-likelihood estimates because, despite their bias, they allow us to do model selection. Then, once we have chosen a model, we can switch to the better REML estimates.

For the analysis of our experimental results, we begin with a model containing all the fixed effects, all interactions between those effects, and no per-subject random effect. We estimate the parameters of the model using maximum likelihood. Then, we add a per-subject random effect, estimate the parameters of the new model, and compare the BIC scores of the two. If the one with the per-subject random effect has a lower BIC (and it does), we adopt it. Then, in a stepwise fashion, we omit each term in the model, re-estimate the parameters, and recalculate the BIC. If we obtain a lower BIC by omitting any of the terms of the model, we drop the term which lowers the BIC the most and repeat the process. When no more terms can be dropped in this way, we adopt the current model as final and estimate the parameters using REML. This procedure is quite typical for mixed-effects model selection [7,16].

As another procedural note, we do not drop terms if they are involved in higher-order interactions that are still part of the model. For instance, we would not drop feature set (F_j) as a factor if the interaction between algorithm and feature set (AF_{hj}) is still in the model. This so-called principle of hierarchy enables us to more easily interpret the resulting model [7]. For the statistical analysis, we used the R statistical programming language (version 2.10.0) [17]. To fit linear mixed-effects models, we used the `lme4` mixed-effects modeling package (version 0.999375-32) [3].

4.2 Results

We begin by describing the equation obtained through model selection since it informs us of the broad relationships between the factors. Then, we present the parameter estimates which quantify the effects of each factor and enable us to make predictions about a detector's future error rates.

Selected model. We arrived at the following model equation to describe the experimental data:

$$\begin{aligned} Y &= \mu + A_h + T_i + U_k + I_l \\ &\quad + AT_{hi} + AU_{hk} + TU_{ik} + UI_{kl} + ATU_{hik} \\ &\quad + S + \epsilon \\ S &\sim N(0, \sigma_s^2) \\ \epsilon &\sim N(0, \sigma_\epsilon^2) \end{aligned} \tag{3}$$

The equation has been split over multiple lines to make it easier to describe. The first line shows the main effects in the model. Note that the algorithm (A_h), training amount (T_i), updating (U_k), and impostor practice (I_l) are all terms retained in the model, but the feature set (F_j) has been dropped. During model selection, it did not substantially improve the fit of the model. This result is not surprising given how little change there was in any single quadrant of Figure 1.

The second line shows the two-way and three-way interaction effects in the model. The interactions between the algorithm, training, and updating (AT_{hi}, AU_{hk}, TU_{ik}, and ATU_{hik}) suggest a complex relationship between these three

factors. The two-way interaction between updating and impostor practice (UI_{kl}) suggests that updating may mitigate the impostor-practice threat. We will explore the nature of these interactions in greater detail when we look at the parameter estimates.

The third line of model equation (3) includes a per-subject random-effect (S) along with the residual error term (ϵ). From the presence of this per-subject term in the model, we conclude that there is substantial typist-to-typist variation. Some typists are easier to distinguish from impostors than others.

Parameter estimates. Table 1 compiles the REML estimates of the parameters. Part (a) provides estimates of all the fixed effects, while Part (b) provides estimates of the random effects. The table is admittedly complex, but we include it for completeness. It contains all the information necessary to make predictions and to approximate the uncertainty of those predictions.

In Part (a), row 1 is the estimate of μ, the equal-error rate for the baseline combination of factor values. The estimate, 17.6%, is the equal-error rate we would predict when operating the Manhattan (scaled) detector with 5 repetitions of training, no updating, and unpracticed impostors.

Rows 2–26 describe how our prediction should change when we depart from the baseline factor values. Each row lists one or more departures along with the estimated change in the predicted equal-error rate. To predict the effect of a change from the baseline values to new values, one would first identify every row in which the new values are listed. Then, one would add the change estimates from each of those rows to the baseline to get a new predicted equal-error rate.

For example, we can predict the equal-error rate for the Nearest Neighbor (Mahalanobis) detector with 200 training repetitions, no updating, and unpracticed impostors. The detector and training amount depart from the baseline configuration. Row 3 lists the effect of switching to Nearest Neighbor (Mahalanobis) as +19.2. Row 6 lists the effect of increasing to 200 training repetitions as −7.2. Row 15 lists the effect of doing both as −18.8 (i.e., an interaction effect). Since the baseline equal-error rate is 17.6%, we would predict the equal-error rate of this new setup to be 10.8% (17.6 + 19.2 − 7.2 − 18.8).

As another example, we can quantify the effect of impostor practice. If no updating strategy is used, according to row 8, impostor practice is predicted to add +1.1 percentage points to a detector's equal-error rate. If a sliding-window updating strategy is used, according to rows 7, 8, and 20, impostor practice is predicted to add +0.7 percentage points (−2.2 + 1.1 + 1.8). Consequently, impostor practice only increases an impostor's chance of success by a percentage point, and the risk is somewhat mitigated by updating.

While the aim of the statistical analysis is to predict what effect each factor has, it would be natural for a practitioner to use these predictions to choose the combination of factor values that give the lowest error. A systematic search of Table 1(a) reveals the lowest predicted equal-error rate to be 7.2%, using the Manhattan (scaled) detector with 100 training repetitions, sliding-window updating, and unpracticed impostors. For very-practiced impostors, the lowest

Table 1. Estimated values for all of the parameters in the model. The estimates are all in percentage points. Part (a) presents the fixed-effects estimates. The first row lists the combination of factor values which comprise the baseline along with the predicted equal-error rate. The remaining 25 rows list changes to the algorithm, amount of training, updating, and impostor practice, along with the predicted change to the equal-error rate. Part (b) presents the estimated typist-to-typist variation and the residual error. Both estimates are expressed as standard deviations rather than variances (i.e., by taking the square-root of the variance estimates) to make them easier to interpret. This table enables us to predict the error rates of a detector under different operating conditions, and also to estimate the uncertainty of those predictions.

		Algorithm	Training	Updating	Impostor	Estimates
1	μ	Manhattan (scaled)	5 reps	None	None	17.6
2	A_h	Outlier-count (z-score)				+2.7
3		Nearest-neighbor (Mahalanobis)				+19.2
4	T_i		50 reps			−4.9
5			100 reps			−6.9
6			200 reps			−7.2
7	U_k			Sliding		−2.2
8	I_l				Very High	+1.1
9	AT_{hi}	Outlier-count (z-score)	50 reps			−1.7
10		Nearest-neighbor (Mahalanobis)	50 reps			−17.0
11		Outlier-count (z-score)	100 reps			−1.3
12		Nearest-neighbor (Mahalanobis)	100 reps			−17.9
13		Outlier-count (z-score)	200 reps			−1.2
14		Nearest-neighbor (Mahalanobis)	200 reps			−18.8
15	AU_{hk}	Outlier-count (z-score)		Sliding		−3.7
16		Nearest-neighbor (Mahalanobis)		Sliding		−7.7
17	TU_{ik}		50 reps	Sliding		−0.8
18			100 reps	Sliding		−1.3
19			200 reps	Sliding		+0.3
20	UI_{kl}			Sliding	Very High	+1.8
21	ATU_{hik}	Outlier-count (z-score)	50 reps	Sliding		+2.3
22		Nearest-neighbor (Mahalanobis)	50 reps	Sliding		+8.1
23		Outlier-count (z-score)	100 reps	Sliding		+4.0
24		Nearest-neighbor (Mahalanobis)	100 reps	Sliding		+7.9
25		Outlier-count (z-score)	200 reps	Sliding		+3.0
26		Nearest-neighbor (Mahalanobis)	200 reps	Sliding		+8.4

(a): Estimates of the fixed-effects parameters of the model (in percentage points).

	Estimates of Standard Deviation
σ_s (Typist-to-typist)	6.0
σ_ϵ (Residual)	6.6

(b): Estimates of the random-effects parameters (as standard deviations).

predicted equal-error rate is 10.1% for the same detector, training amount, and updating strategy.

In Part (b) of Table 1, the typist-to-typist standard deviation (σ_s) is estimated to be 6.0. Since the model assumes the typist-to-typist effect to be Normally distributed, we would predict that about 95% of typists' average equal-error rates will lie withing 2 standard deviations of the values predicted from Part (a). For instance, suppose a new typist were added to a system operating with the baseline factor values. The overall average equal-error rate is predicted to be 17.6%, and with 95% confidence, we would predict that the average equal-error rate for the new typist will be between 5.6% and 29.6% (i.e., $17.6 \pm 2 \times 6.0$).

Likewise, there will be some day-to-day variation in a detector's performance due to unidentified sources of variation. Table 1(b) provides an estimate of 6.6 percentage points for this residual standard deviation (σ_ϵ). Calculations similar to those with the typist-to-typist standard deviation can be used to bound the day-to-day change in detector performance. Note that these confidence intervals are quite large compared to the fixed effects in Part (a). Future research might try to identify the source of the uncertainty and what makes a typist easy or hard to distinguish.

A reader might ask what has really been learned from this statistical analysis. Based on the empirical results in Section 3, it seemed that the Manhattan (scaled) detector with 100 training repetitions and updating had the lowest error. It seemed that the feature set did not matter, and that impostor practice had only a minor effect. To answer, we would say that the statistical analysis has not only supported these observations but explained and enriched them. We now know which factors and interactions are responsible for the low error rate, and we can predict how much that low error rate will change as the typists change and the days progress.

5 Validation

The predictions in the previous section depend on the validity of the model. While the model assumptions can be checked using additional statistical analysis (e.g., residual plots to check Normality), the true test of a model's predictive validity is whether its predictions are accurate. In this section, we describe such a test of validity and the outcome.

5.1 Procedure

We began by replicating the data-collection effort that was used to collect the data described in Section 3. The same apparatus was used to prompt subjects to type the password (`.tie5Roanl`) and to monitor their keystrokes for typing errors. The same high-precision timing device was used to ensure that the keystroke timestamps were collected accurately.

From among the faculty, staff, and students of our university, 15 new subjects were recruited. As before, these subjects typed the password 400 times over 8

sessions of 50 repetitions each. Each session occurred on separate days to capture the natural day-to-day variation in typing rhythms. Their keystroke timestamps were recorded and converted to password-timing vectors, comprised of 11 hold times, 10 keydown-keydown times, and 10 keyup-keydown times.

The evaluation procedure described in Section 3 was replicated using this new data set, for each algorithm, each amount of training data, each feature set, and so on. Each subject was designated as the genuine user in separate evaluations, with the other 14 subjects acting as impostors. We obtained 2,160 ($3 \times 4 \times 3 \times 2 \times 2 \times 15$) new empirical equal-error rates from these evaluations.

To claim that our model is a useful predictor of detector error rates, two properties should hold. First, the difference between the predicted equal-error rate and each subject's average equal-error rate should be Normally distributed with zero mean and variance predicted by the model (σ_s^2). A zero mean indicates that the predicted equal-error rates are correct on average. A Normal distribution of the points around that mean confirms that the typist-to-typist variability has been accurately captured by the model. Second, the residual errors, after the per-subject effects have been taken into account, should also be Normally distributed with zero mean and variance predicted by the model (σ_ϵ^2). This property confirms that the residual variability has also been captured by the model.

To test these properties, we calculate the difference between each empirical equal-error rate and the predicted equal-error rate from the model. This calculation produces 2,160 prediction errors, 144 for each user. The per-typist effect for each user is calculated as the average of these 144 errors. The residual errors are calculated by subtracting the per-typist effect from each prediction error.

5.2 Results

Figure 2 contains two panels, the left one showing the distribution of the per-typist effects, and the right one showing the distribution of the residual errors.

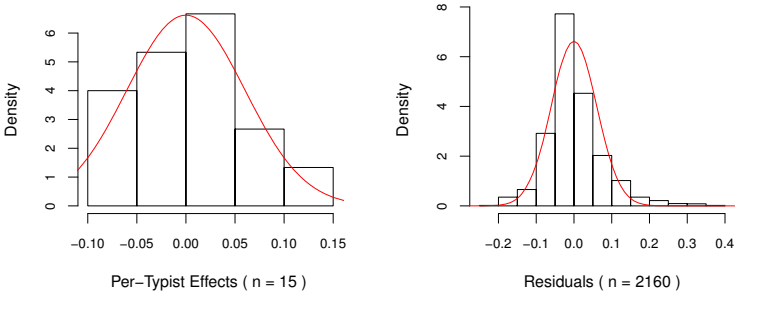


Fig. 2. The distribution of the per-typist effects and the residual errors compared to their predicted Normal distributions. The histogram on the left shows the per-typist effect for the 15 subjects. The histogram on the right depicts the residual errors. Both histograms closely match the Normal distributions predicted by the model. The match between the predicted and the observed distributions validates the model.

Overlaid on each histogram is the Normal distribution predicted by the model. Both the per-typist-effects histogram and the residual-error histogram closely match the predicted Normal distributions.

It is difficult to ascertain Normality from only 15 observations (one per subject), but the per-typist effects appear to be clustered around a mean of zero with the predicted variation about the mean. The residuals appear to be distributed as a bell-shaped curve with a mean of zero and the predicted variance. The tails of the distribution are slightly thicker than Normal, but the overall fit is still very close. Based on these graphs, we conclude that the model can accurately predict the detectors' error rates on a new data set.

6 Related Work

Having demonstrated that we can explain detector's error rates using influential factors from the evaluation, we should put our findings in the context of other keystroke-dynamics research. We cannot review the entire 30 year history of keystroke dynamics, but Peacock et al. [14] provide a concise summary of many of the developments during that time. In this section, we compare our findings to prior research into the influences of the same six factors.

1. **Algorithm:** Several researchers have compared different classification algorithms on a single data set, but few have compared anomaly detectors in this way. Cho et al. [5] compared the Nearest Neighbor (Mahalanobis) detector to an auto-associative neural network. Haider et al. [8] evaluated an Outlier-count (z-score) detector, a different neural network, and a fuzzy-logic detector. In earlier work, we tried to reimplement 14 detectors and replicate the results of earlier evaluations [12]. Our results differed wildly from the earlier ones (e.g., 85.9% vs. 1% error). In fact, the validation we report in Section 5 is one of the few successful replications of error rates from an earlier study using new data.
2. **Amount of training:** Joyce and Gupta [10] used only 8 password repetitions to train their detector, but they found that the same accuracy was observed with as few as 6 repetitions. Araújo et al. [1] considered training sets ranging from 6 to 10 repetitions. To our knowledge, while other researchers have published evaluations using as many as 200 training repetitions [5], no prior work has examined the error rates of detectors trained from as few as 5 to as many as 200 repetitions. In addition, earlier researchers have considered the effect of training on a single algorithm. We found that the effect of training cannot be separated from the effect of the algorithm.
3. **Feature set:** Araújo et al. [1] ran evaluations of a Manhattan (scaled) detector with seven different feature sets, including the three we used. They found that using all three types of feature (e.g., hold times, keydown-keydown times, and keyup-keydown times) produced the best results. In contrast, we found that, so long as hold times and either keydown-keydown times or keyup-keydown times are included, the particular combination has negligible effect.

Our findings benefit from the statistical analysis we used to check whether small effects are substantial enough to be included in the model.

4. **Updating strategy:** Araújo et al. [1] also compared a Manhattan (scaled) detector using updating to one with no updating. Kang et al. [11] compared a k-means detector trained with no updating to ones trained with growing and sliding windows. Both sets of researchers found that an updating strategy lowered error rates. Their results for individual detectors, coupled with our results for three detectors (backed by a validated statistical model), strongly support the claim that updating reduces detector error rates. Our results further demonstrate that window size (i.e., training amount) has an important effect on the error rate of a sliding-window detector.
5. **Impostor practice:** Lee and Cho [13] gave their impostors the opportunity to practice, but they did not describe how many repetitions of practice were taken by each impostor. Araújo et al. [1] split their impostor subjects into two groups. One group observed the genuine user typing the password, and one group did not. The observers seemed to be more successful at mimicking the typing style of the genuine user, but no statistical test was performed. In contrast, our work operationalized practice in terms of the number of repetitions and then quantified the effect of practice on error rates.
6. **Typist-to-typist variation:** To our knowledge, no prior work has substantially investigated whether some typists are easier to distinguish than others (i.e., the typist's effect on detector error rates). Cho et al. [5] implicitly assumed a per-typist effect when they conducted a paired t-test in comparing two detectors. Often, standard deviations are reported along with error rates, but the current work may be the first to explicitly try to understand and quantify the substantial typist-to-typist effect.

Based on our review of the related work, we can make two observations. First, even among those studies that have tried to explain the effects of various factors on detector error rates, interaction effects have not been considered. The present work reveals several complex interactions between algorithm, training, updating, and impostor practice. In the future, we should bear in mind the possibility of interactions between influential factors.

Second, there have been few attempts to generalize from empirical results using statistical analysis. Some researchers have used hypothesis testing (e.g., t-tests) to establish whether there is a statistically significant difference between two sets of error rates [2,5], but such analysis is the rare exception rather than the rule. The current work demonstrates the additional insight and predictive capability that can be gained through statistical analysis. These insights and predictions would be missing if only the raw, empirical error rates were reported.

7 Discussion and Future Work

We initiated the current work to try to explain why different researchers, using the same detectors, were getting wildly different evaluation results. This work has shown that it is possible to replicate earlier results, but great care was taken

to make sure that the details of the replication matched those of the original. We recruited new subjects, but tightly controlled many other factors of the evaluation (e.g., the password, keyboard, and timing mechanism). Showing that results can be replicated is a critical but often under-appreciated part of any scientific discipline. Fortunately, having succeeded, we can begin to vary some of these tightly controlled factors (using the very methodology proposed in this paper), and we can identify which ones threaten replication.

The statistical model presented in this paper is certainly not the last word on which factors influence keystroke-dynamics anomaly detectors. There are factors we did not consider (e.g., the password), and for factors we did consider, there are values we did not (e.g., other anomaly detectors). For the factors and values we did investigate, the rigor of our methodology enables us to make claims with comparatively high confidence. Even knowing that our model is incomplete, and possibly wrong in some respects, it represents a useful guideline which future work can use, test, and refine.[1]

This work is exceptional, as explained in Section 6, for its statistical analysis and validation. Although many anomaly detectors are built upon statistical machine-learning algorithms, statistical analysis is rarely used in the evaluations of those detectors. Often a research paper will propose a new anomaly-detection technique and report the results of a preliminary evaluation. Such evaluations typically show a technique's promise, but rarely provide conclusive evidence of it. As Peisert and Bishop [15] have advocated, researchers must follow preliminary, exploratory evaluations with more rigorous experiments that adhere to the scientific method. Our methodology offers a clear procedure for conducting more rigorous anomaly-detection experiments. We hope it is useful to other researchers.

8 Conclusion

In this work, we aimed to answer two questions. First, what influence do each of six factors—algorithm, training amount, feature set, updating, impostor practice, and typist-to-typist variation—have on keystroke-dynamics error rates? Second, what methodology should we use to establish the effects of these various factors?

In answer to the first question, the detection algorithm, training amount, and updating were found to strongly influence the error rates. We found no difference among our three feature sets, and impostor practice had only a minor effect. Typist-to-typist differences were found to introduce substantial variation; some subjects were much easier to distinguish than others.

In answer to the second question, we proposed a methodology rooted in the scientific method: experimentation, statistical analysis, and validation. This methodology produced a useful, predictive, explanatory model of anomaly-detector error rates. Consequently, we believe that the proposed methodology would add valuable predictive and explanatory power to future anomaly-detection studies.

[1] As George Box notes, "All models are wrong, but some models are useful" [4].

Acknowledgments

The authors are indebted to Howard Seltman and David Banks for sharing their statistical expertise, and to Patricia Loring for running the experiments that provided the data for this paper. We are grateful both to Shing-hon Lau for his helpful comments and also to the anonymous reviewers for theirs.

This work was supported by National Science Foundation grant number CNS-0716677, and by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and W911NF-09-1-0273 from the Army Research Office.

References

1. Araújo, L.C.F., Sucupira, L.H.R., Lizárraga, M.G., Ling, L.L., Yabu-uti, J.B.T.: User authentication through typing biometrics features. IEEE Transactions on Signal Processing 53(2), 851–855 (2005)
2. Bartlow, N., Cukic, B.: Evaluating the reliability of credential hardening through keystroke dynamics. In: Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE 2006), pp. 117–126. IEEE Press, Los Alamitos (2006)
3. Bates, D.: Fitting linear mixed models in R. R. News 5(1), 27–30 (2005)
4. Box, G.E.P., Hunter, J.S., Hunter, W.G.: Statistics for Experimenters: Design, Innovation, and Discovery, 2nd edn. Wiley, New York (2005)
5. Cho, S., Han, C., Han, D.H., Kim, H.I.: Web-based keystroke dynamics identity verification using neural network. Journal of Organizational Computing and Electronic Commerce 10(4), 295–307 (2000)
6. Denning, D.E.: An intrusion-detection model. IEEE Transactions on Software Engineering 13(2) (1987)
7. Faraway, J.J.: Extending Linear Models with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models. Chapman & Hall/CRC (2006)
8. Haider, S., Abbas, A., Zaidi, A.K.: A multi-technique approach for user identification through keystroke dynamics. In: IEEE International Conference on Systems, Man and Cybernetics, pp. 1336–1341 (2000)
9. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. Springer, New York (2001)
10. Joyce, R., Gupta, G.: Identity authentication based on keystroke latencies. Communications of the ACM 33(2), 168–176 (1990)
11. Kang, P., Hwang, S.-s., Cho, S.: Continual retraining of keystroke dynamics based authenticator. In: Lee, S.-W., Li, S.Z. (eds.) ICB 2007. LNCS, vol. 4642, pp. 1203–1211. Springer, Heidelberg (2007)
12. Killourhy, K.S., Maxion, R.A.: Comparing anomaly detectors for keystroke dynamics. In: Proceedings of the 39th Annual International Conference on Dependable Systems and Networks (DSN 2009), June 29-July 2, pp. 125–134. IEEE Computer Society Press, Los Alamitos (2009)
13. Lee, H.j., Cho, S.: Retraining a keystroke dynamics-based authenticator with impostor patterns. Computers & Security 26(4), 300–310 (2007)
14. Peacock, A., Ke, X., Wilkerson, M.: Typing patterns: A key to user identification. IEEE Security and Privacy 2(5), 40–47 (2004)

15. Peisert, S., Bishop, M.: How to design computer security experiments. In: Proceedings of the 5th World Conference on Information Security Education (WISE), pp. 141–148. Springer, New York (2007)
16. Pinheiro, J.C., Bates, D.M.: Mixed-effects Models in S and S-Plus. Statistics and Computing Series. Springer, New York (2000)
17. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2008), http://www.R-project.org
18. Searle, S.R., Casella, G., McCulloch, C.E.: Variance Components. John Wiley & Sons, Inc., Hoboken (2006)
19. Swets, J.A., Pickett, R.M.: Evaluation of Diagnostic Systems: Methods from Signal Detection Theory. Academic Press, New York (1982)

NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring

Paul Giura and Nasir Memon

Polytechnic Intitute of NYU, Six MetroTech Center, Brooklyn, NY

Abstract. With the increasing sophistication of attacks, there is a need for network security monitoring systems that store and examine very large amounts of historical network flow data. An efficient storage infrastructure should provide both high insertion rates and fast data access. Traditional row-oriented Relational Database Management Systems (RDBMS) provide satisfactory query performance for network flow data collected only over a period of several hours. In many cases, such as the detection of sophisticated coordinated attacks, it is crucial to query days, weeks or even months worth of disk resident historical data rapidly. For such monitoring and forensics queries, row oriented databases become I/O bound due to long disk access times. Furthermore, their data insertion rate is proportional to the number of indexes used, and query processing time is increased when it is necessary to load unused attributes along with the used ones. To overcome these problems we propose a new column oriented storage infrastructure for network flow records, called *NetStore*. NetStore is aware of network data semantics and access patterns, and benefits from the simple column oriented layout without the need to meet general purpose RDBMS requirements. The prototype implementation of NetStore can potentially achieve more than ten times query speedup and ninety times less storage size compared to traditional row-stores, while it performs better than existing open source column-stores for network flow data.

1 Introduction

Traditionally intrusion detection systems were designed to detect and flag malicious or suspicious activity in real time. However, such systems are increasingly providing the ability to identify the root cause of a security breach. This may involve checking a suspected host's past network activity, looking up any services run by a host, protocols used, the connection records to other hosts that may or may not be compromised, etc. This requires flexible and fast access to network flow historical data. In this paper we present the design, implementation details and the evaluation of a column-oriented storage infrastructure called *NetStore*, designed to store and analyze very large amounts of network flow data. Throughout this paper we refer to a *flow* as an unidirectional data stream between two endpoints, to a *flow record* as a quantitative description of a flow, and to a *flow ID* as the key that uniquely identifies a flow. In our research the flow ID is

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 277–296, 2010.

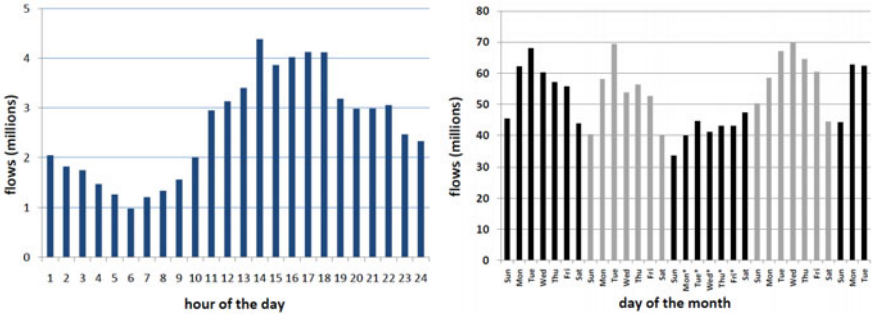


Fig. 1. Flow traffic distribution for one day and one month. In a typical day the busiest time interval is 1PM - 2PM with 4,381,876 flows, and the slowest time interval is 5AM - 6AM with 978,888 flows. For a typical month we noticed the slow down in week-ends and the peek traffic in weekdays. Days marked with * correspond to a break week.

composed of five attributes: *source IP, source port, destination IP, destination port* and *protocol*. We assume that each flow record has associated a start time and an end time representing the time interval when the flow was active in the network.

Challenges. Network flow data can grow very large in the number of records and storage footprint. Figure 1 shows network flow distribution of traffic captured from edge routers in a moderate sized campus network. This network with about 3,000 hosts, commonly reaches up to 1,300 flows/second, an average 53 million flows daily and roughly 1.7 billion flows in a month. We consider records with the average size of 200 Bytes. Besides CISCO NetFlow data [18] there may be other specific information that a sensor can capture from the network such as the IP, transport and application headers information. Hence, in this example, the storage requirement is roughly 10 GB of data per day which adds up to at least 310 GB per month. When working with large amounts of disk resident data, the main challenge is no longer to ensure the necessary storage space, but to minimize the time it takes to process and access the data. An efficient storage and querying infrastructure for network records has to cope with two main technical challenges: keep the insertion rate high, and provide fast access to the desired flow records. When using a traditional row-oriented Relational Database Management Systems (RDBMS), the relevant flow attributes are inserted as a row into a table as they are captured from the network, and are indexed using various techniques [6]. On the one hand, such a system has to establish a trade off between the insertion rate desired and the storage and processing overhead employed by the use of auxiliary indexing data structures. On the other hand, enabling indexing for more attributes ultimately improves query performance but also increases the storage requirements and decreases insertion rates. At query time, all the columns of the table have to be loaded in memory even if only a subset of the attributes are relevant for the query, adding a significant I/O penalty for the overall query processing time by loading unused columns.

When querying disk resident data, an important problem to overcome is the I/O bottleneck caused by large disk to memory data transfers. One potential solution would be to load only data that is relevant to the query. For example, to answer the query "What is the list of all IPs that contacted IP X between dates d_1 and d_2?", the system should load only the source and destination IPs as well as the timestamps of the flows that fall between dates d_1 and d_2. The I/O time can also be decreased if the accessed data is compressed since less data traverses the disk-memory boundary. Further, the overall query response time can be improved if data is processed in compressed format by saving decompression time. Finally, since the system has to insert records at line speed, all the preprocessing algorithms used should add negligible overhead while writing to disk. The above requirements can be met quite well by utilizing a column oriented database as described below.

Column Store. The basic idea of column orientation is to store the data by columns rather than by rows, where each column holds data for a single attribute of the flow and is stored sequentially on disk. Such a strategy makes the system I/O efficient for read queries since only the required attributes related to a query can be read from the disk. The performance benefits of column partitioning were previously analyzed in [9, 2], and some of the ideas were confirmed by the results in the databases academic research community [16, 1, 21] as well as in industry [19, 11, 10, 3]. However, most of commercial and open-source column stores were conceived to follow general purpose RDBMSs requirements, and do not fully use the semantics of the data carried and do not take advantage of the specific types and data access patterns of network forensic and monitoring queries. In this paper we present the design, implementation details and the evaluation of NetStore, a column-oriented storage infrastructure for network records that, unlike the other systems, is intended to provide good performance for network records flow data.

Contribution. The key contributions in this paper include the following:

- Simple and efficient column oriented design of NetStore, a network flow historical storage system that enables quick access to large amounts of data for monitoring and forensic analysis.
- Efficient compression methods and selection strategies to facilitate the best compression for network flow data, that permit accessing and querying data in compressed format.
- Implementation and deployment of NetStore using commodity hardware and open source software as well as analysis and comparison with other open source storage systems used currently in practice.

The rest of this paper is organized as follows: we present related work in Section 2, our system architecture and the details of each component in Section 3. Experimental results and evaluation are presented in Section 4 and we conclude in Section 5.

2 Related Work

The problem of discovering network security incidents has received significant attention over the past years. Most of the work done has focused on near-real time security event detection, by improving existing security mechanisms that monitor traffic at a network perimeter and block known attacks, detect suspicious network behavior such as network scans, or malicious binary transfers [12, 14]. Other systems such as Tribeca [17] and Gigascope [4], use stream databases and process network data as it arrives but do not store the date for retroactive analysis. There has been some work done to store network flow records using a traditional RDBMS such as PostgreSQL [6]. Using this approach, when a NIDS triggers an alarm, the database system builds indexes and materialized views for the attributes that are the subject of the alarm, and could potentially be used by forensics queries in the investigation of the alarm. The system works reasonably well for small networks and is able to help forensic analysis for events that happened over the last few hours. However, queries for traffic spanning more than a few hours become I/O bound and the auxiliary data used to speed up the queries slows down the record insertion process. Therefore, such a solution is not feasible for medium to large networks and not even for small networks in the future, if we consider the accelerated growth of internet traffic. Additionally, a time window of several hours is not a realistic assumption when trying to detect the behavior of a complex botnet engaged in stealthy malicious activity over prolonged periods of time.

In the database community, many researchers have proposed the physical organization of database storage by columns in order to cope with poor read query performance of traditional row-based RDBMS [16, 21, 11, 15, 3]. As shown in [16, 2, 9, 8], a column store provides many times better performance than a row store for read intensive workloads. In [21] the focus is on optimizing the cache-RAM access time by decompressing data in the cache rather than in the RAM. This system assumes the working columns are RAM resident, and shows a performance penalty if data has to be read from the disk and processed in the same run. The solution in [16] relies on processing parallelism by partitioning data into sets of columns, called projections, indexed and sorted together, independent of other projections. This layout has the benefit of rapid loading of the attributes belonging to the same projection and referred to by the same query without the use of auxiliary data structure for tuple reconstruction. However, when attributes from different projections are accessed, the tuple reconstruction process adds significant overhead to the data access pattern. The system presented in [15] emphasizes the use of an auxiliary metadata layer on top of the column partitioning that is shown to be an efficient alternative to the indexing approach. However, the metadata overhead is sizable and the design does not take into account the correlation between various attributes.

Finally, in [9] authors present several factors that should be considered when one has to decide to use a column store versus a row store for a read intensive workload. The relative large number of network flow attributes and the workloads

with the predominant set of queries with large selectivity and few predicates favor the use of a column store system for historical network flow records storage.

NetStore is a column oriented storage infrastructure that shares some of the features with the other systems, and is designed to provide the best performance for large amounts of disk resident network flow records. It avoids tuple reconstruction overhead by keeping at all times the same order of elements in all columns. It provides fast data insertion and quick querying by dynamically choosing the most suitable compression method available and using a simple and efficient design with a negligible meta data layer overhead.

3 Architecture

In this section we describe the architecture and the key components of NetStore. We first present the characteristics of network data and query types that guide our design. We then describe the technical design details: how the data is partitioned into columns, how columns are partitioned into segments, what are the compression methods used and how a compression method is selected for each segment. We finally present the metadata associated with each segment, the *index nodes*, and the internal IPs inverted index structure, as well as the basic set of operators.

3.1 Network Flow Data

Network flow records and the queries made on them show some special characteristics compared to other time sequential data, and we tried to apply this knowledge as early as possible in the design of the system. First, flow attributes tend to exhibit temporal clustering, that is, the range of values is small within short time intervals. Second, the attributes of the flows with the same source IP and destination IP tend to have the same values (e.g. port numbers, protocols, packets sizes etc.). Third, columns of some attributes can be efficiently encoded when partitioned into time based segments that are encoded independently. Finally, most attributes that are of interest for monitoring and forensics can be encoded using basic integer data types.

The records insertion operation is represented by bulk loads of time sequential data that will not be updated after writing. Having the attributes stored in the same order across the columns makes the join operation become trivial when attributes from more than one column are used together. Network data analysis does not require fast random access on all the attributes. Most of the monitoring queries need fast sequential access to large number of records and the ability to aggregate and summarize the data over a time window. Forensic queries access specific predictable attributes but collected over longer periods of time. To observe their specific characteristics we first compiled a comprehensive list of forensic and monitoring queries used in practice in various scenarios [5]. Based on the data access pattern, we identified five types among the initial list. *Spot* queries (S) that target a single key (usually an IP address or port number)

and return a list with the values associated with that key. *Range* queries (R) that return a list with results for multiple keys (usually attributes corresponding to the IPs of a subnet). *Aggregation* queries (A) that aggregate the data for the entire network and return the result of the aggregation (e.g. traffic sent out for network). *Spot Aggregation* queries (SA) that aggregate the values found for one key in a single value. *Range Aggregation* queries (RA) that aggregate data for multiple keys into a single value. Examples of these types of queries expressed in plain words:

(S) "What applications are observed on host X between dates d_1 and d_2?"
(R) "What is the list of destination IPs that have source IPs in a subnet between dates d_1 and d_2?"
(A) "What is the total number of connections for the entire network between dates d_1 and d_2?"
(SA) "What is the number of bytes that host X sent between dates d_1 and d_2?"
(RA) "What is the number of hosts that each of the hosts in a subnet contacted between dates d_1 and d_2?"

3.2 Column Oriented Storage

Columns. In NetStore, we consider that flow records with n attributes are stored in the logical table with n columns and an increasing number of rows (tuples) one for each flow record. The values of each attribute are stored in one column and have the same data type. By default almost all of the values of a column are not sorted. Having the data sorted in a column might help get better compression and faster retrieval, but changing the initial order of the elements requires the use of auxiliary data structure for tuple reconstruction at query time. We investigated several techniques to ease tuple reconstruction and all methods added much more overhead at query time than the benefit of better compression and faster data access. Therefore, we decided to maintain the same order of elements across columns to avoid any tuple reconstruction penalty when querying. However, since we can afford one column to be sorted without the need to use any reconstruction auxiliary data, we choose to first sort only one column and partially sort the rest of the columns. We call the first sorted column the *anchor* column. Note that after sorting, given our storage architecture, each segment can still be processed independently. The main purpose of the anchor column choosing algorithm is to select the ordering that facilitates the best compression and fast data access. Network flow data express strong correlation between several attributes and we exploit this characteristic by keeping the strongly correlated columns in consecutive sorting order as much as possible for better compression results. Additionally, based on previous queries data access pattern, columns are arranged by taking into account the probability of each column to be accessed by future queries. The columns with higher probabilities are arranged at the beginning of the sorting order. As such, we maintain the counting probabilities associated with each of the columns given by the formula $P(c_i) = \frac{a_i}{t}$, where c_i is the i-th column, a_i the number of queries that accessed c_i and t the total number of queries.

Segments. Each column is further partitioned into fixed sets of values called *segments*. Segments partitioning enables physical storage and processing at a smaller granularity than simple column based partitioning. These design decisions provide more flexibility for compression strategies and data access. At query time only used segments will be read from disk and processed based on the information collected from segments metadata structures called *index nodes*. Each segment has associated a unique identifier called *segment ID*. For each column, a segment ID represents an auto incremental number, started at the installation of the system. The segment sizes are dependent of the hardware configuration and can be set in such a way to use the most of available main memory. For better control over data structures used, the segments have the same number of values across all the columns. In this way there is no need to store a record ID for each value of a segment, and this is one major difference compared to some existing column stores [11]. As we will show in Section 4 the performance of the system is related to the segment size used. The larger the segment size, the better the compression performance and query processing times. However, we notice that records insertion speed decreases with the increase of segment size, so, there is a trade off between the query performance desired and the insertion speed needed. Most of the columns store segments in compressed format and, in a later section we present the compression algorithms used. Column segmentation design is an important difference compared to traditional row oriented systems that process data a tuple at a time, whereas NetStore processes data segment at a time, which translates to many tuples at a time. Figure 3 shows the processing steps for the three processing phases: buffering, segmenting and query processing.

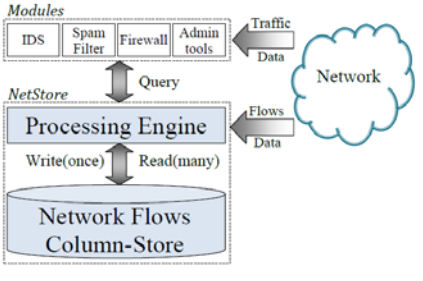


Fig. 2. NetStore main components: Processing Engine and Column-Store.

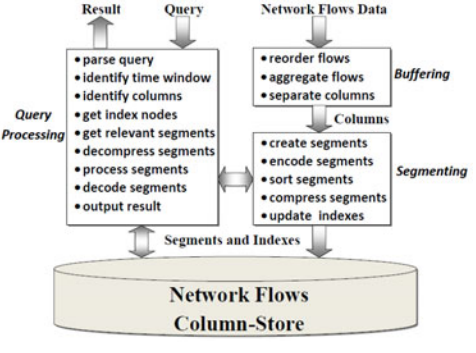


Fig. 3. NetStore processing phases: buffering, segmenting and query processing.

Column Index. For each column we store the meta data associated with each of the segments in an *index node* corresponding to the segment. The set of all index nodes for the segments of a column represent the *column index*. The information in each index node includes statistics about data and different features that are used in the decision about the compression method to use and optimal data

access, as well as the time interval associated with the segment in the format $[min\_start\_time, max\_end\_time]$. Figure 4 presents an intuitive representation of the columns, segments and index for each column. Each column index is implemented using a time interval tree. Every query is relative to a time window T. At query time, the index of every column accessed is looked up and only the segments that have the time interval overlapping window T are considered for processing. In the next step, the statistics on segment values are checked to decide if the segment should be loaded in memory and decompressed. This two-phase index processing helps in early filtering out unused data in query processing similar to what is done in [15]. Note that the index nodes do not hold data values, but statistics about the segments such as the minimum and the maximum values, the time interval of the segment, the compression method used, the number of distinct values, etc. Therefore, index usage adds negligible storage and processing overhead.

From the list of initial queries we observed that the column for the source IP attribute is most frequently accessed. Therefore, we choose this column as our first sorted anchor column, and used it as a clustered index for each source IP segment. However, for workloads where the predominant query types are spot queries targeting a specific column other than the anchor column, the use of indexes for values inside the column segments is beneficial at a cost of increased storage and slowdown in insertion rate. Thus, this situation can be acceptable for slow networks were the insertion rate requirements are not too high. When the insertion rate is high then it is best not to use any index but rely on the meta-data from the index nodes.

Internal IPs Index. Besides the column index, NetStore maintains another indexing data structure for the network internal IP addresses called the *Internal IPs index*. Essentially the IPs index is an inverted index for the internal IPs. That is, for each internal IP address the index stores in a list the absolute positions where the IP address occurs in the column, $sourceIP$ or $destIP$, as if the column is not partitioned into segments. Figure 5 shows an intuitive representation of the IPs index. For each internal IP address the positions list represents an array of increasing integer values that are compressed and stored on disk on a daily basis. Because IP addresses tend to occur in consecutive positions in a column, we chose to compress the positions list by applying run-length-encoding on differences between adjacent values.

3.3 Compression

Each of the segments in NetStore is compressed independently. We observed that segments within a column did not have the same distribution due to the temporal variation of network activity in working hours, days, nights, weekends, breaks etc. Hence segments of the same column were best compressed using different methods. We explored different compression methods. We investigated methods that allow data processing in compressed format and do not need decompression of all the segment values if only one value is requested. We also looked at methods

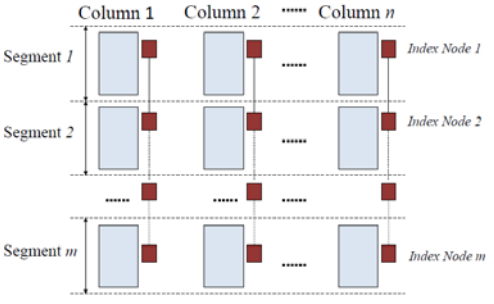


Fig. 4. Schematic representation of columns, segments, index nodes and column indexes

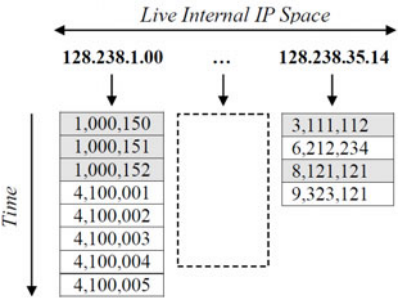


Fig. 5. Intuitive representation of the IPs inverted index

that provide fast decompression and reasonable compression ratio and speed. The decision on which compression algorithm to use is done automatically for each segment, and is based on the data features of the segment such as data type, the number of distinct values, range of the values and number of switches between adjacent values. We tested a wide range of compression methods, including some we designed for the purpose or currently used by similar systems in [1,16,21,11], with needed variations if any. Below we list the techniques that emerged effective based on our experimentation:

- **Run-Length Encoding (RLE):** is used for segments that have few distinct repetitive values. If value v appears consecutively r times, and $r > 1$, we compress it as the pair (v, r). It provides fast compression as well as the ability to process data in compressed format.
- **Variable Byte Encoding:** is a byte-oriented encoding method used for positive integers. It uses a variable number of bytes to encode each integer value as follows: if $value < 128$ use one byte (set highest bit to 0), for $value < 128 * 128$ use 2 bytes (first byte has highest bit set to 1 and second to 0) and so on. This method can be used in conjunction with RLE for both values and runs. It provides reasonable compression ratio and good decompression speed allowing the decompression of only the requested value without the need to decompress the whole segment.
- **Dictionary Encoding:** is used for columns with few distinct values and sometimes before RLE is applied (e.g. to encode "protocol" attribute).
- **Frame Of Reference:** considers the interval bounded by the minimum and maximum values as the frame of reference for the values to be compressed [7]. We use it to compress non-empty timestamp attributes within a segment (e.g. start time, end time, etc.) that are integer values representing the number of seconds from the epoch. Typically the time difference between minimum and maximum timestamp values in a segment is less than few hours, therefore the encoding of the difference is possible using short values of 2 bytes instead of integers of 4 bytes. It allows processing data in compressed format by decompressing each timestamp value individually without the need to decompress the whole segment.

- **Generic Compression:** we use the DEFLATE algorithm from the zlib library that is a variation of the LZ77 [20]. This method provides compression at the binary level, and does not allow values to be individually accessed unless the whole segment is decompressed. It is chosen if it enables faster data insertion and access than the value-based methods presented earlier.
- **No Compression:** is listed as a compression method since it will represent the base case for our compression selection algorithm.

Method Selection. The selection of a compression method is done based on the statistics collected in one pass over the data of each segment. As mentioned earlier, the two major requirements of our system are to keep records insertion rates high and to provide fast data access. Data compression does not always provide better insertion and better query performance compared to "No compression", and for this we developed a model to decide on when compression is suitable and if so, what method to choose. Essentially, we compute a score for each candidate compression method and we select the one that has the best score. More formally, we assume we have $k+1$ compression methods $m_0, m_1, \ldots, m_k$, with m_0 being the "No Compression" method. We then compute the insertion time as the time to compress and write to disk, and the access time, to read from disk and decompress, as functions of each compression method. For value-based compression methods, we estimate the compression, write, read and decompression times based on the statistics collected for each segment. For the generic compression we estimate the parameters based on the average results obtained when processing sample segments. For each segment we evaluate:

$$\begin{aligned} insertion\,(m_i) &= c\,(m_i) + w\,(m_i)\,, \quad i = 1, \ldots, k \\ access\,(m_i) &= r\,(m_i) + d\,(m_i)\,, \quad i = 1, \ldots, k \end{aligned}$$

As the base case for each method evaluation we consider the "No Compression" method. We take I_0 to represent the time to insert an uncompressed segment which is represented by only the writing time since there is no time spent for compression and, similarly A_0 to represent the time to access the segment which is represented by only the time to read the segment from disk since there is no decompression. Formally, following the above equations we have:

$$insertion\,(m_0) = w\,(m_0) = I_0 \text{ and } access\,(m_0) = r\,(m_0) = A_0$$

We then choose the candidate compression methods m_i only if we have both:

$$insertion\,(m_i) < I_0 \text{ and } access\,(m_i) < A_0$$

Next, among the candidate compression methods we choose the one that provides the lowest access time. Note that we primarily consider the access time as the main differentiator factor and not the insertion time. The disk read is the most frequent and time consuming operation and it is many times slower than disk write of the same size file for commodity hard drives. Additionally, insertion time can be improved by bulk loading or by other ways that take into account that the network traffic rate is not steady and varies greatly over time,

whereas the access mechanism should provide the same level of performance at all times.

The model presented above does not take into account if the data can be processed in compressed format and the assumption is that decompression is necessary at all times. However, for a more accurate compression method selection we should include the probability of a query processing the data in compressed format in the access time equation. Since forensic and monitoring queries are usually predictable, we can assume without affecting the generality of our system, that we have a total number of t queries, each query q_j having the probability of occurrence p_j with $\sum_{j=1}^{t} p_j = 1$. We consider the probability of a segment s being processed in compressed format as the probability of occurrence of the queries that process the segment in compressed format. Let CF be the set of all the queries that process s in compressed format, we then get:

$$P(s) = \sum_{q_j \in CF} p_j, \quad CF = \{q_j | q_j \text{ processes } s \text{ in compressed format}\}$$

Now, a more accurate access time equation can be rewritten taking into account the possibility of not decompressing the segment for each access:

$$access(m_i) = r(m_i) + d(m_i) \cdot (1 - P(s)), \quad i = 1, \ldots, k \tag{1}$$

Note that the compression selection model can accommodate any compression, not only the ones mentioned in this paper, and is also valid in the cases when the probability of processing the data in compressed format is 0.

3.4 Query Processing

Figure 3 illustrates NetStore data flow, from network flow record insertion to the query result output. Data is written only once in bulk, and read many times for processing. NetStore does not support transaction processing queries such as record updates or deletes, it is suitable for analytical queries in general and network forensics and monitoring queries in special.

Data Insertion. Network data is processed in several phases before being delivered to permanent storage. First, raw flow data is collected from the network sensors and is then preprocessed. Preprocessing includes the *buffering* and *segmenting* phases. Each flow is identified by a flow ID represented by the 5-tuple [*sourceIP, sourcePort, destIP, destPort, protocol*]. In the buffering phase, raw network flow information is collected until the buffer is filled. The flow records in the buffer are aggregated and then sorted. As mentioned in Section 3.3, the purpose of sorting is twofold: better compression and faster data access. All the columns are sorted following the sorting order determined based on access probabilities and correlation between columns using the first sorted column as anchor.

In the segmenting phase, all the columns are partitioned into segments, that is, once the number of flow records reach the buffer capacity the column data in the buffer is considered a full segment and is processed. Each of the segments is then compressed using the appropriate compression method based on the data it carries. The information about the compression method used and statistics about the data is collected and stored in the index node associated with the segment. Note that once the segments are created, the statistics collection and compression of each segment is done independent of the rest of the segments in the same column or in other columns. By doing so, the system takes advantage of the increasing number of cores in a machine and provides good record insertion rates in multi threaded environments.

After preprocessing all the data is sent to permanent storage. As monitoring queries tend to access the most recent data, some data is also kept in memory for a predefined length of time. NetStore uses a small active window of size W and all the requests from queries accessing the data in the time interval [NOW - W, NOW] are served from memory, where NOW represents the actual time of the query.

Query Execution. For flexibility NetStore supports limited SQL syntax and implements a basic set of segment operators related to the query types presented in Section 3.1. Each SQL query statement is translated into a statement in terms of the basic set of segment operators. Below we briefly present each general operator:

- **filter_segs** (d_1, d_2): Returns the set with segment IDs of the segments that overlap with the time interval $[d_1, d_2]$. This operator is used by all queries.
- **filter_atts**$(segIDs, pred_1(att_1), \ldots, pred_k(att_k))$: Returns the list of pairs $(segID, pos\_list)$, where pos_list represents the intersection of attribute position lists in the corresponding segment with id $segID$, for which the attribute att_i satisfies the predicate $pred_i$, with $i = 1, \ldots, k$.
- **aggregate** $(segIDs, pred_1(att_1), \ldots, pred_k(att_k))$: Returns the result of aggregating values of attribute att_k by att_{k-1} by $\ldots$ att_1 that satisfy their corresponding predicates $pred_k, \ldots, pred_1$ in segments with ids in $segIDs$. The aggregation can be summation, counting, min or max.

The queries considered in section 3.1 can all be expressed in terms of the above operators. For example the query: "What is the number of unique hosts that each of the hosts in the network contacted in the interval $[d_1, d_2]$?" can be expressed as follows: **aggregate**(**filter_segs**(d_1, d_2), $sourceIP = 128.238.0.0/16$, $destIP$). After the operator **filter_segs** is applied, only the $sourceIP$ and $destIP$ segments that overlap with the time interval $[d_1, d_2]$ are considered for processing and their corresponding index nodes are read from disk. Since this is a range aggregation query, all the considered segments will be loaded and processed. If we consider the query "What is the number of unique hosts that host X contacted in the interval $[d_1, d_2]$?" it can be expressed as follows: **aggregate**(**filter_segs**(d_1, d_2), $sourceIP$ = X, $destIP$). For this query the number of relevant segments can be reduced even more by discarding the ones that do

not overlap with the time interval $[d_1, d_2]$, as well as the ones that don't hold the value X for *sourceIP* by checking corresponding index nodes statistics. If the value X represents the IP address of an internal node, then the internal IPs index will be used to retrieve all the positions where the value X occurs in the *sourceIP* column. Then a count operation is performed of all the unique *destIP* addresses corresponding to the positions. Note that by using internal IPs index, the data of *sourceIP* column is not touched. The only information loaded in memory is the positions list of IP X as well as the segments in column *destIP* that correspond to those positions.

4 Evaluation

In this section we present an evaluation of NetStore. We designed and implemented NetStore using the Java programming language on the FreeBSD 7.2-RELEASE platform. For all the experiments we used a single machine with 6 GB DDR2 RAM, two Quad-Core 2.3 Ghz CPUs, 1TB SATA-300 32 MB Buffer 7200 rpm disk with a RAID-Z configuration. We consider this machine representative of what a medium scale enterprise will use as a storage server for network flow records.

For experiments we used the network flow data captured over a 24 hour period of one weekday at our campus border router. The size of raw text file data was about 8 GB, 62,397,593 network flow records. For our experiments we considered only 12 attributes for each network flow record, that is only the ones that were meaningful for the queries presented in this paper. Table 1 shows the attributes used as well as the types and the size for each attribute. We compared NetStore's performance with two open source RDBMS, a row-store, PostgreSQL [13] and a column-store, LucidDB [11]. We chose PostgreSQL over other open source systems because we intended to follow the example in [6] which uses it for similar tasks. Additionally we intended to make use of the partial index support for internal IPs that other systems don't offer in order to compare the performance of our inverted IPs index. We chose LucidDB as the column-store to compare with as it is, to the best of our knowledge, the only stable open source column-store that yields good performance for disk resident data and provides reasonable insertion speed. We chose only data captured over one day, with size slightly larger than the available memory, because we wanted to maintain reasonable running times for the other systems that we compared NetStore to. These systems become very slow for larger data sets and performance gap compared to NetStore increases with the size of the data.

4.1 Parameters

Figure 6 shows the influence that the segment size has over the insertion rate. We observe that the insertion rate drops with the increase of segment size. This trend is expected and is caused by the delay in preprocessing phase, mostly because of the larger segment array sorting. As Figure 7 shows, the segment

Table 1. NetStore flow attributes.

Column	Type	Bytes
sourceIP	int	4
destIP	int	4
sourcePort	short	2
destPort	short	2
protocol	byte	1
startTime	short	2
endTime	short	2
tcpSyns	byte	1
tcpAcks	byte	1
tcpFins	byte	1
tcpRsts	byte	1
numBytes	int	4

Table 2. NetStore properties and network rates supported based on 24 hour flow records data and the 12 attributes

Property	Value	Unit
records insertion rate	10,000	records/second
number of records	62,397,594	records
number of bytes transported	1.17	Terabytes
bytes transported per record	20,616.64	Bytes/record
bits rate supported	1.54	Gbit/s
number of packets transported	2,028,392,356	packets
packets transported per record	32.51	packets/record
packets rate supported	325,075.41	packets/second

size also affects the compression ratio of each segment, the larger the segment size the larger the compression ratio achieved. But high compression ratio is not a critical requirement. The size of the segments is more critically related to the available memory, the desired insertion rate for the network and the number of attributes used for each record. We set the insertion rate goal at 10,000 records/second, and for this goal we set a segment size of 2 million records given the above hardware specification and records sizes. Table 2 shows the insertion performance of NetStore. The numbers presented are computed based on average bytes per record and average packets per record given the insertion rate of 10,000 records/second. When installed on a machine with the above specification, NetStore can keep up with traffic rates up to 1.5 Gbit/s for the current experimental implementation. For a constant memory size, this rate decreases with the increase in segment size and the increase in the number of attributes for each flow record.

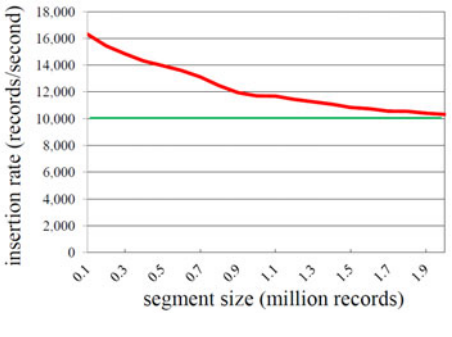


Fig. 6. Insertion rate for different segment sizes

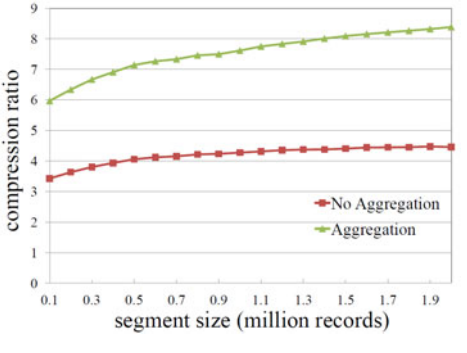


Fig. 7. Compression ratio with and without aggregation

4.2 Queries

Having described the NetStore architecture and it's design details, in this section we consider the queries described in [5], but taking into account data collected over the 24 hours for internal network 128.238.0.0/16. We consider both the queries and methodology in [5] meaningful for how an investigator will perform security analysis on network flow data. We assume all the flow attributes used are inserted into a table *flow* and we use standard SQL to describe all our examples.

Scanning. Scanning attack refers to the activity of sending a large number of TCP SYN packets to a wide range of IP addresses. Based on the received answer the attacker can determine if a particular vulnerable service is running on the victim's host. As such, we want to identify any TCP SYN scanning activity initiated by an external hosts, with no TCP ACK or TCP FIN flags set and targeted against a large number of internal IP destinations, larger than a preset limit. We use the following range aggregation query (Q1):

```
SELECT sourceIP, destPort, count(distinct destIP), startTime
FROM flow
WHERE sourceIP <> 128.238.0.0/16 AND destIP = 128.238.0.0/16
AND protocol = tcp AND tcpSyns = 1 AND tcpAcks = 0 AND tcpFins = 0
GROUP BY sourceIP
HAVING count(distinct destIP) > limit;
```

External IP address 61.139.105.163 was found scanning starting at time t_1. We check if there were any valid responses after time t_1 from the internal hosts, where no packet had the TCP RST flag set, and we use the following query (Q2):

```
SELECT sourceIP, sourcePort, destIP
FROM flow
WHERE startTime > t1 AND sourceIP = 128.238.0.0/16
AND destIP = 61.139.105.163 AND protocol = tcp AND tcpRsts = 0;
```

Worm Infected Hosts. Internal host with the IP address 128.238.1.100 was discovered to have been responded to a scanning initiated by a host infected with the Conficker worm and we want to check if the internal host is compromised. Typically, after a host is infected, the worm copies itself into memory and begins propagating to random IP addresses across a network by exploiting the same vulnerability. The worm opens a random port and starts scanning random IPs on port 445. We use the following query to check the internal host (Q3):

```
SELECT sourceIP, destPort, count(distinct destIP)
FROM flow
WHERE startTime > t1 AND sourceIP = 128.238.1.100 AND destPort = 445;
```

SYN Flooding. It is a network based-denial of service attack in which the attacker sends an unusual large number of SYN request, over a threshold t, to a specific target over a small time window W. To detect such an attack we filter all the incoming traffic and count the number of flows with TCP SYN bit set and no TCP ACK or TCP FIN for all the internal hosts. We use the following query(Q4):

```
SELECT destIP, count(distinct sourceP), startTime
FROM flow
WHERE startTime > 'NOW - W' AND destIP = 128.238.0.0/16
AND protocol = tcp AND tcpSyns = 1 AND tcpAcks = 0 AND tcpFins = 0
GROUP BY destIP
HAVING count(sourceIP) > t;
```

Network Statistics. Besides security analysis, network statistics and performance monitoring is another important usage for network flow data. To get this information we use aggregation queries for all collected data over a large time window, both incoming and outgoing. Aggregation operation can be number of bytes or packets summation, number of unique hosts contacted or some other meaningful aggregation statistics. For example we use the following simple aggregation query to find the number of bytes transported in the last 24 hours (Q5):

```
SELECT sum(numBytes)
FROM flow WHERE startTime > 'NOW - 24h';
```

General Queries. The sample queries described above are complex and belong to more than one basic type described in Section 3.1. However, each of them can be separated into several basic types such that the result of one query becomes the input for the next one. We build a more general set of queries starting from the ones described above by varying the parameters in such a way to achieve different level of data selectivity form low to high. Then, for each type we reported the average performance for all the queries of that type. Figure 8 shows the average running times of selected queries for increasing segment sizes. We observe that for S type queries that don't use IPs index (e.g. for attributes other than internal sourceIP or destIP), the performance decreases when the segment size increases. This is an expected result since for larger segments there is more unused data loaded as part of the segment where the spotted value resides. When using the IPs index the performance benefit comes from skipping the irrelevant segments whose positions are not found in the positions list. However, for internal busy servers that have corresponding flow records in all the segments, all corresponding segments of attributes have to be read but not the IPs segments. This is an advantage since an IP segment is several times larger in general than the other attributes segments. Hence, except for spot queries that use non-indexed attributes, queries tend to be faster for larger segment sizes.

4.3 Compression

Our goal with using compression is not to achieve the best compression ratio nor the best compression or decompression speed, but to obtain the highest records

insertion rate and the best query performance. We evaluated our compression selection model by comparing performance when using a single method for all the segments in the column, with the performance when using the compression selection algorithm for each segment. To select the method for a column we compressed first all the segments of the columns with all the six methods presented. We then measured the access performance for each column compressed with each method. Finally, we selected the compression method of a column, the method that provides the best access times for the majority of the segments.

For the variable segments compression, we activated the methods selection mechanism for all columns and then we inserted the data, compressing each segment based on the statistics of its own data rather than the entire column. In both cases we did not change anything in the statistic collection process since all the statistics were used in the query process for both approaches. We obtained on an average 10 to 15 percent improvement per query using the segment based compression method selection model with no penalty for the insertion rate. However, we consider the overall performance of compression methods selection model is satisfactory and the true value resides in the framework implementation, being limited only by the individual methods used not by the general model design. If the data changes and other compression methods are more efficient for the new data, only the compression algorithm and the operators that work on this compressed data should be changed, with the overall architecture remaining the same. Some commercial systems [19] apply on top of the value-based compressed columns another layer of general binary compression for increased performance. We investigated the same possibility and compared four different approaches to compression on top of the implemented column oriented architecture: no compression, value-based compression only, binary compression only and value-based plus binary compression on top of that. For the no compression case, we processed the data using the same indexing structure and column oriented layout but with the compression disabled for all the segments. For the binary compression only we compress each segment using the generic binary compression. In the case of value-based compression we compress all the segments having the dynamic selection mechanism enabled, and for the last approach we apply another layer of generic compression on top of already value-based compressed segments.

The results of our experiment for the four cases are shown in Figure 9. We can see that compression is a determining factor in performance metrics. Using value-based compression achieves the best average running time for the queries while the uncompressed segments scenario yields the worst performance.We also see that adding another compression layer does not help in query performance nor in the insertion rate even though it provides better compression ratio. However, the general compression method can be used for data aging, to compress and archive older data that is not actively used.

Figure 7 shows the compression performance for different segment sizes and how flow aggregation affects storage footprint. As expected, compression performance is better for larger segment sizes in both cases, with and without aggregation. That is the case because of the compression methods used. The larger the

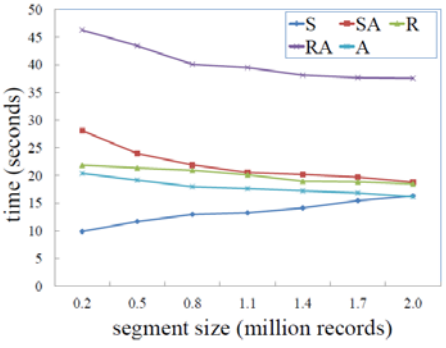


Fig. 8. Average query times for different segment sizes and different query types

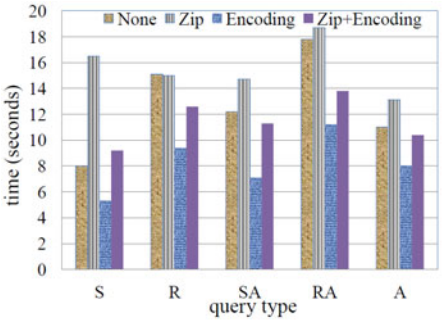


Fig. 9. Average query times for the compression strategies implemented

segment, the longer the runs for column with few distinct values, the smaller the dictionary size for each segment. The overall compression ratio of raw network flow data for the segment size of 2 million records is 4.5 with no aggregation and 8.4 with aggregation enabled. Note that the size of compressed data includes also the size of both indexing structures: column indexes and IPs index.

4.4 Comparison with Other Systems

For comparison we used the same data and performed a system-specific tuning for each of the systems parameters. To maintain the insertion rate above our target of 10,000 records/second we created three indexes for each Postgres and Luciddb: one clustered index on startTime and two un-clustered indexes, one on sourceIP and one on destIP attributes. Although we believe we chose good values for the other tuning parameters we cannot guarantee they are optimal and we only present the performance we observed. We show the performance for using the data and the example queries presented in Section 4.2.

Table 3 shows the relative performance of NetStore compared to PostgresSQL for the same data. Since our main goal is to improve disk resident data access, we ran each query once for each system to minimize the use of cached data. The numbers presented show how many times NetStore is better.

To maintain a fair overall comparison we created a PostgresSQL table for each column of Netstore. As mentioned in [2], row-stores with columnar design provide better performance for queries that access a small number of columns such as the sample queries in Section 4.2. We observe that Netstore clearly outperforms

Table 3. Relative performance of NetStore versus columns only PostgreSQL and LucidDB for query running times and total storage needed

	Q1	Q2	Q3	Q4	Q5	Storage
Postgres/NetStore	10.98	7.98	2.21	15.46	1.67	93.6
LucidDB/NetStore	5.14	1.10	2.25	2.58	1.53	6.04

PostgreSQL for all the query types providing the best results for queries accessing more attributes (e.g. Q1 and Q4) even though it uses 90 times more disk space including all the auxiliary data. The poor PostgreSQL performance can be explained by the absence of more clustered indexes, the lack of compression, and the unnecessary tuple overhead.

Table 3 also shows the relative performance compared to LucidDB. We observe that the performance gap is not at the same order of magnitude compared to that of PostgreSQL even when more attributes are accessed. However, NetStore performs clearly better when storing about 6 times less data. The performance penalty of LucidDB can be explain by the lack of column segmentation design and by early materialization in the processing phase specific to general-purpose column stores. However we noticed that LucidDB achieves a significant performance improvement for the subsequent runs of the same query by efficiently using memory resident data.

5 Conclusion and Future Work

With the growth of network traffic, there is an increasing demand for solutions to better manage and take advantage of the wealth of network flow information recorded for monitoring and forensic investigations. The problem is no longer the availability and the storage capacity of the data, but the ability to quickly extract the relevant information about potential malicious activities that can affect network security and resources. In this paper we have presented the design, implementation and evaluation of a novel working architecture, called NetStore, that is useful in the network monitoring tasks and assists in network forensics investigations.

The simple column oriented design of NetStore helps in reducing query processing time by spending less time for disk I/O and loading only needed data. The column partitioning facilitates the use of efficient compression methods for network flow attributes that allow data processing in compressed format, therefore boosting query runtime performance. NetStore clearly outperforms existing row-based DBMSs systems and provides better results that the general purpose column oriented systems because of simple design decisions tailored for network flow records. Experiments show that NetStore can provide more than ten times faster query response compared to other storage systems while maintaining much smaller storage size. In future work we seek to explore the use of NetStore for new types of time sequential data, such as host log analysis, and the possibility to release it as an open source system.

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: SIGMOD 2006: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 671–682. ACM, New York (2006)

2. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In: SIGMOD 2008: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 967–980. ACM, New York (2008)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2006 (2006)
4. Cranor, C., Johnson, T., Spataschek, O., Shkapenyuk, V.: Gigascope: a stream database for network applications. In: SIGMOD 2003: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 647–651. ACM, New York (2003)
5. Gates, C., Collins, M., Duggan, M., Kompanek, A., Thomas, M.: More netflow tools for performance and security. In: LISA 2004: Proceedings of the 18th USENIX Conference on System Administration, pp. 121–132. USENIX Association, Berkeley (2004)
6. Geambasu, R., Bragin, T., Jung, J., Balazinska, M.: On-demand view materialization and indexing for network forensic analysis. In: NETB 2007: Proceedings of the 3rd USENIX International Workshop on Networking Meets Databases, pp. 1–7. USENIX Association, Berkeley (2007)
7. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: Proceedings of IEEE International Conference on Data Engineering, pp. 370–379 (1998)
8. Halverson, A., Beckmann, J.L., Naughton, J.F., Dewitt, D.J.: A comparison of c-store and row-store in a common framework. Technical Report TR1570, University of Wisconsin-Madison (2006)
9. Holloway, A.L., DeWitt, D.J.: Read-optimized databases, in depth. Proc. VLDB Endow. 1(1), 502–513 (2008)
10. Infobright Inc. Infobright, `http://www.infobright.com`
11. LucidEra. Luciddb, `http://www.luciddb.org`
12. Paxson, V.: Bro: A system for detecting network intruders in real-time. Computer Networks, 2435–2463 (1998)
13. PostgreSQL. Postgresql, `http://www.postgresql.org`
14. Roesch, M.: Snort - lightweight intrusion detection for networks. In: LISA 1999: Proceedings of the 13th USENIX Conference on System Administration, pp. 229–238. USENIX Association, Berkeley (1999)
15. Ślęzak, D., Wróblewski, J., Eastwood, V., Synak, P.: Brighthouse: an analytic data warehouse for ad-hoc queries. Proc. VLDB Endow. 1(2), 1337–1345 (2008)
16. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented dbms. In: VLDB 2005: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB Endowment, pp. 553–564 (2005)
17. Sullivan, M., Heybey, A.: Tribeca: A system for managing large databases of network traffic. In: USENIX, pp. 13–24 (1998)
18. Cisco Systems. Cisco ios netflow, `http://www.cisco.com`
19. Vertica Systems. Vertica, `http://www.vertica.com`
20. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23, 337–343 (1977)
21. Zukowski, M., Boncz, P.A., Nes, N., Héman, S.: Monetdb/x100 - a dbms in the cpu cache. IEEE Data Eng. Bull. 28(2), 17–22 (2005)

Live and Trustworthy Forensic Analysis of Commodity Production Systems

Lorenzo Martignoni[1], Aristide Fattori[2],
Roberto Paleari[2], and Lorenzo Cavallaro[3]

[1] Università degli Studi di Udine, Italy
lorenzo.martignoni@uniud.it
[2] Università degli Studi di Milano, Italy
{aristide,roberto}@security.dico.unimi.it
[3] Vrije Universiteit Amsterdam, The Netherlands
sullivan@few.vu.nl

Abstract. We present HyperSleuth, a framework that leverages the virtualization extensions provided by commodity hardware to securely perform live forensic analysis of potentially compromised production systems. HyperSleuth provides a trusted execution environment that guarantees four fundamental properties. First, an attacker controlling the system cannot interfere with the analysis and cannot tamper the results. Second, the framework can be installed as the system runs, without a reboot and without loosing any volatile data. Third, the analysis performed is completely transparent to the OS and to an attacker. Finally, the analysis can be periodically and safely interrupted to resume normal execution of the system. On top of HyperSleuth we implemented three forensic analysis applications: a lazy physical memory dumper, a lie detector, and a system call tracer. The experimental evaluation we conducted demonstrated that even time consuming analysis, such as the dump of the content of the physical memory, can be securely performed without interrupting the services offered by the system.

1 Introduction

Kernel-level malware, which compromise the kernel of an operating system (OS), are one of the most important concerns systems security experts have to fight with, nowadays [1]. Being executed at the same privilege level of the OS, such a malware can easily fool traditional analysis and detection techniques. For instance, Shadow Walker exploits kernel-level privileges to defeat memory content scanners by providing a de-synchronized view of the memory used by the malware and the one perceived by the detector [2].

To address the problem of kernel-level malware and of attackers that are able to obtain kernel-level privileges, researchers proposed to run *out-of-the-box* analyses by exploiting virtual machine monitor (VMM), or hypervisor, technology. In such a context, the analysis is executed in a trusted environment, the VMM, while the monitored OS and users' applications, are run as a guest of the virtual machine. Recently, this research direction has been strongly encouraged by the introduction of hardware extensions for the x86 architecture that simplify the

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 297–316, 2010.

development of virtual machine monitors [3,4]. Since the hypervisor operates at a higher privilege level than the guest OS, it has complete control of the hardware, it can preemptively intercept events, it cannot be tampered by a compromised OS, and therefore it can be used to enforce stronger protection [5, 6, 7, 8, 9]. Advanced techniques, like the one used by Shadow Walker to hide malicious code, are defeated using out-of-the-box memory content scanners. Unfortunately, all the VMM-based solutions proposed in literature are based on the same assumption: they operate *proactively*. In other words, the hypervisor must be started before the guest OS and it must run until the guest terminates. Therefore, post-infection analysis of systems that were not running such VMM-based protections before an infection continues to be unsafe, because the malware and the tools used for the analysis run at the same privilege level.

In this paper we propose HyperSleuth, a tool that exploits the VMM extensions available nowadays (and typically unused) in commodity hardware, to securely perform *live forensic analyses* of potentially compromised production systems. HyperSleuth is executed on systems that are believed to be compromised, and obtains complete and tamper-resistant control over the OS, by running at "ring minus-one" (i.e., the hypervisor privilege level). HyperSleuth consists in (i) a tiny hypervisor that performs the analysis and (ii) a secure loader that installs the hypervisor and verifies that its code is not tampered during installation. Like in virtualization-based malware, the *hypervisor is installed on-the-fly*: the alleged compromised host OS is transformed into a guest as it runs [10]. Since the hardware guarantees that the hypervisor is not accessible from the guest code, HyperSleuth remains persistent in the system for all the time necessary to perform the live analysis. On the contrary, other solutions proposed in literature for executing verified code in untrusted environments are not persistent and thus cannot guarantee that the verified code is not tampered when the execution of the untrusted code is resumed [11, 12, 13]. By providing a persistent trusted execution environment, HyperSleuth opens new opportunities for live and trusted forensic analyses, including the possibility to perform analyses that require to monitor the run-time behavior of the system. When the live analysis is concluded positively (e.g., no malicious program is found), HyperSleuth can be removed from the system and the OS, which was temporarily transformed into a guest OS, becomes again the host OS. As for the installation, the hypervisor is removed on-the-fly.

We developed a memory acquisition tool, a lie detector [6], and a system call tracer on top of HyperSleuth, to show how our hardware-supported VMM-based framework can be successfully used to gather volatile data even from production systems whose services cannot be interrupted. To experimentally demonstrate our claims about the effectiveness of HyperSleuth, we simulated two scenarios: a compromised production system running a heavy-loaded DNS server and a system infected by several kernel-level malware. We used HyperSleuth to dump the content of the physical memory of the former and to detect the malware in the latter. In the first case, HyperSleuth was able to dump the entire content of the physical memory, without interrupting the services offered by the server. In the second case, HyperSleuth detected all the infections.

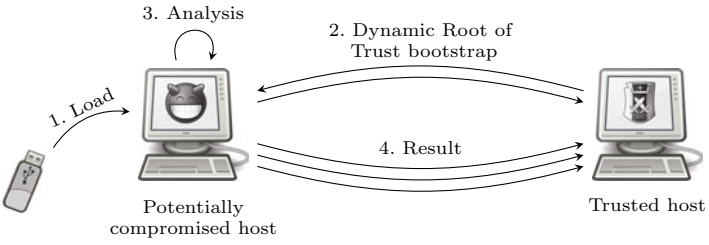


Fig. 1. Overview of HyperSleuth execution

2 Overview

HyperSleuth should not be considered merely as a forensic tool, but rather as a framework for constructing forensic tools. Indeed, its goal is to provide a trusted execution environment for performing any *live forensic analysis* on production systems. More precisely, the execution environment in which a forensic analysis should be performed must guarantee four fundamental properties. First, the environment must guarantee a *tamper-proof* execution of the analysis code. That is, an attacker controlling the system cannot interfere with the analysis and cannot tamper the results. Second, it must be possible to perform an *a-posteriori bootstrap* of the trusted execution environment, even after the system has been compromised, and the bootstrap process itself must require no specific support from the system. Third, the trusted execution environment must be completely *transparent* to the system and to the attacker. Fourth, the trusted execution environment must be *persistent*. That is, the analysis performed in the trusted environment can be periodically interrupted, and the normal execution of the system resumed. Practically speaking, that allows to analyze an alleged compromised system without freezing it and without interrupting the services it provides. Moreover, such a property would allow to perform forensic analyses that require to monitor the run-time behavior of the system. As we will briefly see in the next sections, HyperSleuth fulfills all the aforementioned properties and can thus be used to safely analyze any compromised system that meets the requirements described in Section 2.3.

Figure 1 depicts the execution of HyperSleuth. HyperSleuth is installed and executed on demand (step 1 in Figure 1), only when there is a suspect that the host has been compromised, or in general when there is the necessity to perform a live forensic analysis. The execution is characterized by two phases. In the first phase (step 2 in Figure 1), HyperSleuth assumes complete control of the host and establishes a Dynamic Root of Trust (DRT). That is accomplished with the collaboration of a trusted host (located in the same local network). The trusted host is responsible for attesting that the DRT has been correctly established. In the second phase (steps 3–4 in Figure 1), HyperSleuth performs a specific live forensic analysis and transmits the results of the analysis to the trusted host. Since the trusted host has a proof that the DRT has been correctly established and since, in turn, the DRT guarantees that the analysis code executes in the

untrusted host untampered, the results of the analysis can be transitively considered authentic.

In the following, we briefly describe the architecture of HyperSleuth and how it manages to assume and maintain complete control of the untrusted host. Then, we describe the mechanism we use to bootstrap the dynamic root of trust, and, finally, we describe the assumptions and the threat model under which HyperSleuth runs.

2.1 HyperSleuth Architecture

HyperSleuth needs to be isolated from the host OS, to prevent any attack potentially originating from a compromised system. Simultaneously, HyperSleuth must be able to access certain resources of the host, to perform the requested forensic analysis, and to access the network to transmit the result to the trusted machine.

Figure 2 shows the position where HyperSleuth resides in the host. Since HyperSleuth needs to obtain and maintain complete control of the host and needs to operate with more privileges than the attacker, it resides at the lowest level: between the hardware and the host OS. HyperSleuth exploits hardware virtualization support available in commodity x86 CPUs [3, 4] (which is typically unused). In other words, it executes at the privilege level of a Virtual Machine Monitor (VMM) and thus it has direct access to the hardware and its isolation from the host OS is facilitated by the CPU.

One of the peculiar features of HyperSleuth is the possibility to load and unload the VMM as the host runs. This hot-plug capability is indeed a very important feature: it allows to transparently take over an allegedly compromised system, turning, *on-the-fly*, its host OS into a guest one, and vice-versa at will. This is done without rebooting the system and thus preserving all those valuable runtime information that can allow to discover a malware infection or an intrusion. To do that, HyperSleuth leverages a characteristic of the hardware virtualization support available in x86 CPUs that allows to launch a VMM at any time, even when the host OS and users' applications are already running. Once the VMM is launched, the host becomes a *guest* of the VMM and the attacker loses her monopoly of the system and any possibility to tamper the execution of the VMM and the results of the forensic analysis.

The greyed portions in Figure 2 represent the trusted components in our system. During the launch, HyperSleuth assumes complete control of virtual memory management, to ensure that the host OS cannot access any of its private memory locations. Moreover, HyperSleuth does not trust any existing software component of the host. Rather, it contains all the necessary primitives to inspect directly the state of the guest and to dialog with the network card to transmit data to the trusted party.

Depending on the type of forensic analysis, the analysis might be performed immediately after the launch, or it might be executed in multiple rounds, interleaved with the execution of the OS and users' applications. The advantage of the latter approach over the former is that the host can continue its normal

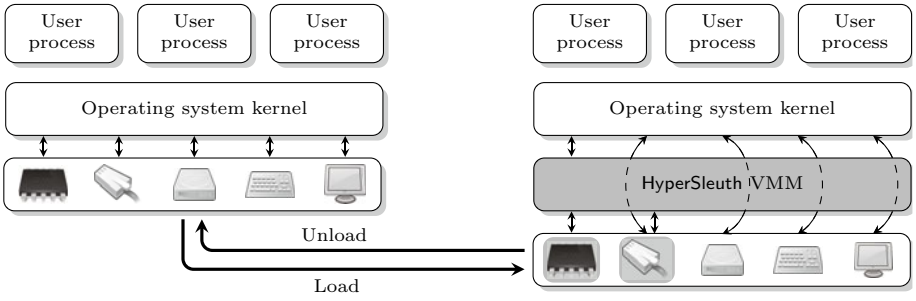


Fig. 2. Overview of HyperSleuth architecture

activity while the analysis is being performed. Thus, the analysis does not result in a denial of service and can also target run-time evolving characteristics of the system. In both cases, when the analysis is completed, HyperSleuth can be disabled and even unloaded.

2.2 HyperSleuth Trusted Launch

HyperSleuth's launch process consists in enabling the VMM privilege level, in configuring the CPU to execute HyperSleuth code at this level, and in configuring the CPU such that all virtual memory management operations can be intercepted and supervised by the VMM. Unfortunately, an attacker could easily tamper the launch. For example, she could simulate a successful installation of the VMM and then transmit fake analysis results to the trusted host. This weakness stems from the fact that the launch process just described lacks an initial trusted component on which we can rely to establish the DRT.

The approach we use to establish the DRT is based on a primitive for tamper proof code execution. This primitive allows to create and to prove the establishment of a minimalistic trusted execution environment that guarantees that the code executed in this environment runs with maximum available privileges and that no attacker can manipulate the code before and during the execution. We use this primitive to create the environment to launch HyperSleuth and to prove to the trusted host that we have established the missing trusted component and that all subsequent operations are secured.

We currently rely on a pure software primitive that is based on a challenge and response protocol and involves an external trusted host [14]. Alternatively, a TPM-based hardware attestation primitive can be used for this purpose (e.g., Intel `senter` and AMD `skinit` primitives [3, 15]).

2.3 Requirements and Threat Model

Since HyperSleuth leverages hardware support for virtualization available in commodity CPUs, such support must be available on the system that must be

analyzed[1]. To maximize the portability of HyperSleuth, we have designed it to only require first generation of hardware facilities for virtualization (i.e., HyperSleuth does not require extensions for MMU and I/O virtualization). Clearly, HyperSleuth cannot be used on systems on which virtualization support is already in use [16]. If a trusted VMM were already running on the host, the VMM could be used directly to perform the analysis. On the other side, if a malicious VMM were running on the host, HyperSleuth's trusted launch would fail.

In order to launch HyperSleuth some privileged instructions must be executed. That can be accomplished by installing a kernel driver in the target host. Note that, in the unlikely case of a damaged system that does not allow to load any kernel driver, alternative solutions for executing code in the kernel can be used (e.g., the page-file attack [10]).

The threat model under which HyperSleuth operates takes into consideration a very powerful attacker, e.g., an attacker with kernel-level privileges. Nonetheless, some assumptions were made while designing HyperSleuth. In particular, the attacker does not operate in system management mode, the attacker does not perform hardware-based attacks (e.g., a DMA-based attack), and the attacker does not leverage an external and more powerful host to simulate the bootstrap of the DRT. Some of these assumptions could indeed be relaxed by virtualizing completely I/O devices using either a pure-software approach or recent hardware support for devices virtualization (e.g., Intel VT-d), and by employing an hardware trusted platform for code attestation (e.g., TPM), keeping HyperSleuth a secure and powerful framework for performing forensic analysis of live data.

3 Implementation

The core of HyperSleuth is a minimalistic virtual machine monitor that is installed on the host while the OS and users' applications are already running. We achieve this goal by exploiting hardware support for virtualization available in modern x86 CPUs. In this Section we describe how we have implemented HyperSleuth on a system with an Intel x86 CPU with VT-x extensions.

3.1 Intel VT-x

Before presenting the details of HyperSleuth VMM implementation, we give a brief overview of the hardware virtualization technology available in Intel x86 CPUs, called VT-x. AMD technology, named SVM, is very similar and differs mostly in terms of terminology.

Intel VT-x separates the CPU execution into two modes of operation: *VMX root mode* and *VMX non-root mode*. The VMM and the guest (OS and applications) execute respectively in root and non-root modes. Software executing in both modes can operate in any of the four privilege levels that are supported

[1] Although nowadays all consumer CPUs come with hardware support for virtualization, in order to be usable, the support must be enabled via the BIOS. At the moment we do not know how many manufactures enable the support by default.

by the CPU. Thus, the guest OS can execute at the highest CPU privilege and the VMM can supervise the execution of the guest without any modification of the guest. When a VMM is installed, the CPU switches back and forth between non-root and root mode: the execution of the guest might be interrupted by an *exit* to root mode and subsequently resumed by an *enter* to non-root mode. After the launch, the VMM execution is never scheduled and exits to root-mode are the only mechanism for the VMM to regain the control of the execution. Like hardware exceptions, exits are events that block the execution of the guest, switch from non-root mode to root mode, and transfer the control to the VMM. However, differently from exceptions, the set of events triggering exits to root mode can be configured dynamically by the VMM. Examples of exiting events are exceptions, interrupts, I/O operations, and the execution of privileged instructions that access control registers or descriptor tables. Exits can also be requested explicitly by the guest through a *VMM call*. Exits are handled by a specific VMM routine that eventually executes an enter to resume the execution of the guest. The state of the CPU at the time of an exit and of an enter is stored in a data structure called Virtual Machine Control Structure, or *VMCS*. This structure also controls the set of events triggering exists and the state of the CPU for executing in root-mode.

In the typical deployment, the launch of the VMM consists of three steps. First, the VMX root-mode is enabled. Second, the CPU is configured to execute the VMM in root-mode. Third, the guests are booted in non-root mode. However, Intel VT-x allows to launch a VMM at any time, thus giving the ability to transform a running host into a guest of a VMM. The procedure for such a delayed launch is the same as the one just described, with the exception of the third step. The state of the CPU for non-root mode is set to the exact same state of the CPU preceding the launch, such that, when the launch is completed, the execution of the OS and its applications resumes in non-root mode. The inverse procedure can be used to unload the VMM, disable VMX root-mode, and give back full control of the system to the OS.

3.2 HyperSleuth VMM

HyperSleuth can be loaded at any time by exploiting the delayed launch feature offered by the CPU. Figure 3 shows a simplified memory layout after the launch of HyperSleuth. The environment for non-root mode, in which the OS and users' application are executed, is left intact. The environment for root mode instead is created during the launch and maintained isolated by the VMM. The VMCS controls the execution contexts of both root and non-root modes. In the following paragraphs we describe in details the steps required to launch the VMM, to recreate the environment for running the OS and users' applications, and to enforce the isolation of root-mode from non-root mode.

VMM Launch. To launch HyperSleuth VMM in a running host we perform the following operations. First, we allocate a fixed-size chunk of memory to hold the data and code of the VMM. Second, we enable VMX root-mode. Third, we

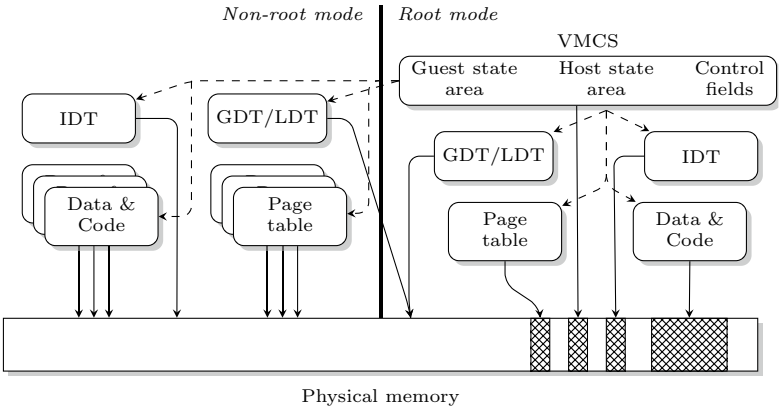


Fig. 3. Memory layout after the launch of HyperSleuth; ⇢ denotes the CPU contexts stored in the VMCS, ⟶ denotes physical memory mappings, and ▩ denotes the physical memory locations of the VMM that must not be made accessible to the guest.

create and initialize the VMCS. Fourth, we resume the normal execution of the guest by entering non-root mode.

When, at the end of the launch, the CPU enters non-root mode, it loads the context for executing the guest from the *guest-state area* of the VMCS. The trick to load the VMM without interrupting the execution of the OS and users' applications is to set, in the VMCS, the context for non-root mode to the same context in which the launch was initiated. The context in which the VMM executes is instead defined by the *host-state area* of the VMCS. Like during an enter, the CPU loads the context from the VMCS during an exit. The context is created from scratch during the launch and the host-state area is configured accordingly. In particular, we create and register a dummy Interrupt Descriptor Table (to ignore interrupts that might occur during switches between the two VMX modes), we register the Global and Local Descriptor Tables (we use the same tables used in non-root mode), we register the address of the VMM entry point (i.e., the address of the routine for handling exits), and we assign the stack.

The set of events that trigger exits to root-mode are defined in the *execution control fields* of the VMCS. The configuration of these fields depends on the type of the forensic analysis we want to perform and can be changed dynamically.

VMM Trusted Launch. Although on the paper the launch of the VMM appears a very simple process, it requires to perform several operations. Such operations must be performed atomically, otherwise a skilled attacker may interfere with the whole bootstrap process and tamper VMM code and data. To maximize HyperSleuth portability, we decided to address this problem using a software-based primitive for tamper-proof code execution. The primitive we rely on is thoroughly described in [14]. In a few words, the primitive is based on a challenge-response protocol and a checksum function. The trusted host issues a challenge for the untrusted system and the challenge consists in computing a

checksum. The result of the checksum is sent back to the trusted host. A valid checksum received within a predefined time is the proof that a Trusted Computing Base (TCB) has been established on the untrusted system. The checksum function is constructed such that the correct checksum value can be computed in time only if the checksum function and the code for launching the VMM are not tampered, and if the environment in which the checksum is computed and in which the VMM launch will be performed guarantees that no attacker can interrupt the execution and regain the control of the execution before the launch is completed. Practically speaking, the correct checksum will be computed in time only if the computation and the launch are performed with kernel privileges, with interrupts disabled, and no VMM is running.

MMU Virtualization. In order to guarantee complete isolation of the VMM from the guest, it is essential to ensure that the guest cannot access any of the memory pages in use by the VMM (i.e., the crosshatched regions in Figure 3). However, to perform any useful analysis, we need the opposite to be possible.

Although modern x86 CPUs provide hardware support for MMU virtualization, we have opted for a software-based approach to maximize the portability of HyperSleuth. The approach we use is based on the assumption that the direct access to physical memory locations is not allowed by the CPU (with paging enabled) and that physical memory locations are referenced through virtual addresses. The CPU maintains a mapping between virtual and physical memory locations and manages the permissions of these locations through page tables. By assuming the complete control of the page tables, the VMM can decide which physical locations the guest can access. To do that, the VMM maintains a *shadow page table* for each page table used by the guest, and tricks the guest into using the shadow page table instead of the real one [17].

A shadow page table is a clone of the original page table and is used to maintain a different mapping between virtual and host physical addresses and to enforce stricter memory protections. In our particular scenario, where the VMM manages a single guest and the OS has already filled the page tables (because the VMM launch is delayed), the specific duty of the shadow page table is to maintain as much as possible the original mapping between virtual and physical addresses and to ensure that none of the pages assigned to the VMM is mapped into a virtual page accessible to the guest. As described in Section 4, we also rely on the shadow page table to restrict and trap certain memory accesses to perform the live forensic analysis. The algorithm we currently use to maintain the shadow page tables trades off performance for simplicity and is based on tracing and simulating all accesses to tables.

Unrestricted Guest Access to I/O Devices. In the typical deployment, physical I/O devices connected to the host are shared between the VMM and one or more guests. In our particular scenario, instead, there is no need to share any I/O device between the guest and the VMM: HyperSleuth executes batch and interacts only with the trusted host via network. Thus, the guest can be given direct and unrestricted access to I/O devices. Since the OS runs in non-root mode, unmodified, and at the highest privilege level, it is authorized to perform

I/O operations, unless the VMM configures the execution control fields of the VMCS such that I/O operations cause exits to root-mode. By not doing so, the VMM allows the guest OS to perform unrestricted and direct I/O. This approach simplifies drastically the architecture of the VMM and, most importantly, allows the OS to continue to perform I/O activities exactly as before, without any additional overhead.

Direct Network Access. HyperSleuth relies on a trusted host to bootstrap the dynamic root of trust and to store the result of the analysis. Since we are assuming that no existing software component of the host can be trusted, the only viable approach to communicate securely over the network is to dialog directly with the network card. For this reason, HyperSleuth contains a minimalistic network driver that supports the card available on the host. All the data transmitted over the network is encapsulated in UDP packets. Packets are signed and encrypted automatically by the driver using a pre-shared key, which we hardcode in HyperSleuth just before the launch.

As described in the previous paragraph, HyperSleuth does not virtualize hardware peripherals, but it lets the guest to access them directly. Thus, the network card must be shared transparently with the guest. In other words, to avoid interferences with the network activity of the guest, HyperSleuth must save and restore the original state of the card (i.e., the content of PCI registers), respectively before and after using the network. To transmit a packet the driver writes the physical address and the size of the packet to the appropriate control registers of the device. The driver then polls the status register of the device until the transmission is completed. Polling is used because, for simplicity, we execute all VMM code with interrupts disabled. Packets reception is implemented in the same way.

VMM Removal. HyperSleuth can be completely removed from the system at the end of the analysis. The removal essentially is the opposite process of the launch. First, we disable VMX root-mode. Second, we deallocate the memory regions assigned to the VMM (e.g., the Interrupt Descriptor Table, the stack, and the code). Third, we update the context of the CPU such that the OS and users' applications can resume their normal execution. More precisely, we set the context to that stored in the guest-state area of the VMCS, which reflects the context of the CPU in non-root mode when the last exit occurred. Fourth, we transfer the execution to a small snippet of code that deallocates the VMCS and then transfers the control to where the execution was interrupted in non-root mode.

4 Live Forensic Analysis

HyperSleuth operates completely in batch mode. The only user action required is to copy an executable on the system to be analyzed and to fire its execution. This executable is a loader that establishes the dynamic root of trust by creating a tamper-proof execution environment and by using this environment to launch

the VMM. Note that, the loader is removed from the memory and the disk to prevent malicious software to detect its presence. Once launched, the VMM performs the forensic analysis, transmits the results to the trusted hosts and then removes itself.

Although HyperSleuth VMM is completely transparent to the OS and users' applications and it is removed after the end of the analysis, the launch of the VMM is a slightly invasive process. Indeed, it requires to execute the loader that in turn loads a kernel driver (to launch the VMM) and might start other additional in-guest utilities. Our claim is that, considered the valuable volatile information HyperSleuth can gather from the system, the little modifications its installation produces to the state of the system are an acceptable compromise. After all, no zero invasive solution for *a posteriori* forensic analysis exists.

Currently, HyperSleuth supports three live forensic applications: a lazy physical memory dumper, a lie detector, and a system call tracer. Clearly, all these analyses could be performed also without the need of a dynamic root of trust and the VMM. Indeed, there are several commercial and open source applications with the same capabilities available, but, by operating at the same privilege level of the OS kernel to analyze, they can easily be tampered by an attacker (with the same privileges), and cannot thus provide the safety guarantees offered by HyperSleuth.

4.1 Physical Memory Dumper

Traditional approaches for dumping the content of the physical memory are typically based on kernel drivers or on FireWire devices. Unfortunately, both approaches have a major drawback that limits their applicability to non production systems. Dumping the content of the physical memory is an operation that should be performed atomically, to guarantee the integrity of the dumped data. Failing to achieve this would, in fact, enable an attacker to make arbitrary modification to the content of the memory, potentially hampering any forensic analysis of live data. On the other side, if the dump is performed atomically, the system, and the services the system provides, will be blocked for the entire duration of the dump. That is not desirable, especially if there is only a marginal evidence that the system has been compromised. Being the dump very time consuming, the downtime might be economically very expensive and even dangerous.

To address this problem, we exploit HyperSleuth's persistent trusted execution environment to implement a new approach for dumping lazily the content of the physical memory. This approach guarantees that the state of the physical memory dumped corresponds to the state of the memory *at the time the dump is requested*. That is, no malicious process can "clean" the memory after HyperSleuth has been installed. Moreover, being performed lazily, the dump of the state of the memory does not monopolize the CPU and does not interrupt the execution of the processes running in the system. In other words, HyperSleuth allows to dump the content of the physical memory even of a production system without causing any outage of the services offered by the system.

```
1   switch (VMM exit reason)
2     case CR3 write:
3        Sync PT and SPT
4        for (v = 0; v < sizeof(SPT); v++)
5           if (SPT[v].Writable && !DUMPED[SPT[v].PhysicalAddress])
6              SPT[v].Writable = 0;
7
8     case Page fault: // 'v' is the faulty address
9        if (PT/SPT access)
10          Sync PT and SPT and protect SPTEs if necessary
11       else if (write access && PT[v].Writable)
12          if (!DUMPED[PT[v].PhysicalAddress])
13             DUMP(PT[v].PhysicalAddress);
14          SPT[v].Writable = DUMPED[PT[v].PhysicalAddress] = 1;
15       else
16          Pass the exception to the OS
17
18    case Hlt:
19       for (p = 0; p < sizeof(DUMPED); p++)
20          if (!DUMPED[p])
21             DUMP(p); DUMPED[p] = 1;
22             break;
```

Fig. 4. Algorithm for lazy dump of the physical memory

The dump of the memory is transmitted via network to the trusted host. Each page is fragmented, to fit the MTU of the channel, and labelled. The receiver reassembles the fragments and reorders the pages to reconstruct the original bitstream image of the physical memory. To ease further analysis, the image produced by HyperSleuth is compatible with off-the-shelf tools for memory forensic analysis (e.g., Volatility [18]).

The algorithm we developed for dumping lazily the content of the physical memory is partially inspired by the technique used by operating systems for handling shared memory and known as *copy-on-write*. The rationale of the algorithm is that the dump of a physical memory page can be safely postponed until the page is accessed for writing. More precisely, the algorithm adopts a combination of two strategies to dump the memory: *dump-on-write* (DOW), and *dump-on-idle* (DOI). The former permits to dump a page before it is modified by the guest; the latter permits to dump a page when the guest is idle. Note that the algorithm assumes that the guest cannot access directly the physical memory. However, an attacker could still program a hardware device to alter the content of the memory by performing a DMA operation. In our current threat model we do not consider DMA-based attacks.

Figure 4 shows the pseudo-code of our memory dumper. Essentially the VMM intercepts three types of events: updates of the page table address, page-fault exceptions, and CPU idle loops. The algorithm maintains a map of the physical pages that have already been dumped (`DUMPED`) and leverages the shadow page table (`SPT`) to enforce stricter permissions than the ones specified in the real page table (`PT`) currently used by the system. When the page table address (stored in the `CR3` register) is updated, typically during a context switch, the algorithm synchronizes the shadow page table and the page table (line 3). Subsequently, all the entries of the shadow page table mapping physical not yet dumped pages

are granted read-only permissions (lines 4–6). Such a protection ensures that all the memory accesses performed by the guest OS for writing to any virtual page mapped into a physical page that has not been dumped yet results in a page fault exception. The VMM intercepts all the page fault exceptions for keeping the shadow page table and the real page table in sync, for reinforcing our write protection after every update of the page table (lines 9–10), and also for intercepting all write accesses to pages not yet dumped (lines 11–14). The latter type of faults are characterized by a write access to a non-writable virtual page that is marked as writable in the real page table. If the accessed physical page has not been dumped yet, the algorithm dumps the page and flags it as such. All other types of page fault exceptions are delivered to the guest OS that will manage them accordingly. Finally, the VMM detects CPU idle loops by intercepting all occurrences of the `hlt` instruction. This instruction is executed by the OS when there is no immediate work to be done, and it halts the CPU until an interrupt is delivered. We exploit these short idle periods to dump the pending pages (lines 19–22). It is worth noting that a loaded system might enter very few idle loops. For this reason, at every context switch we check whether the CPU has recently entered the idle loop and, if not, we force a dump of a small subset of the pending pages (not shown in the figure).

4.2 Lie Detector

Kernel-level malware are particularly insidious as they operate at a very high privilege level and can, in principle, hide any resource an attacker wants to protect from being discovered (e.g., processes, network communications, files). Different techniques exist to achieve such a goal (see [1]), but all of them aim at forcing the OS to lie about its state, eventually. Therefore, the only effective way to discover such liars is to compare the state of the system perceived from the system itself with the state of the system perceived by a VMM. Unfortunately, so far lie detection has been possible only using a traditional VMM and thus it has not been applicable on production systems not already deployed in virtual machine environments. On the other hand, HyperSleuth's hot-plug capability of securely migrating a host OS into a guest one (and vice-versa) on-the-fly makes it a perfect candidate for detecting liars in production systems that had not been deployed in virtual machine environments since the beginning.

To this end, besides launching the VMM, HyperSleuth loader runs a simple in-guest utility that collects detailed information about the state of the system and transmits its output to the trusted host. This utility performs the operations typically performed by system tools to display information about the state of the system and intentionally relies on the untrusted code of the OS. The intent is to trigger the malicious code installed by the attacker to hide any malicious software component or activity. For example, this utility collects the list of running processes, active networks connections, loaded drivers, open files and registry keys, and so on. At the end of its execution, the utility performs a VMM call to transfer the execution to the HyperSleuth VMM. At this point the VMM collects the same information through OS-aware inspection. That is, the

VMM does not rely on any untrusted code of the system, but rather implements its own primitives for inspecting the state of the guest and, when possible, offers multiple primitives to inspect the state of the same resource. For example it offers primitives to retrieve the list of running processes/threads, each of which relies on a different data structure available in the kernel. Finally, the trusted host compares the views provided by the in-guest utility and the VMM.

Since the state of the system changes dynamically and since the in-guest utility and the VMM does not run simultaneously, we repeat the procedure multiple times, with a variable delay between each run to limit any measurement error.

4.3 System Call Tracer

System calls tracing has been widely recognized as a way to infer, observe, and understand the behavior of processes [19]. Traditionally, system calls were invoked by executing software interrupt instructions causing a transition from user-space to kernel-space. Such user-/kernel-space interactions can be intercepted by HyperSleuth, as interrupt instructions executed by the guest OS in VMX non-root mode cause an exit to VMX root mode, i.e., to the VMM.

Alternative and more efficient mechanisms for user-/kernel-space interactions have been introduced by CPU developers, recently. Unfortunately, Intel VT-x does not support natively the tracing of system calls invoked through the `sysenter/sysexit` fast invocation interface used by modern operating systems. The approach we use to trace system calls is thus inspired by Ether [5]. System calls are intercepted through another type of exits: synthetic page fault exceptions. All system calls invocations go through a common gate, whose address is defined in the `SYSENTER_EIP` register. We shadow the value of this register and set the value of the shadow copy to the address of a non-existent memory location, such that all system calls invocations result in a page fault exception and in an exit to root mode. The VMM can easily detect the reason of the fault by inspecting the faulty address. When a system call invocation is trapped by the VMM, it logs the system call and then resumes the execution of the guest from the real address of `SYSENTER_EIP`. To intercept returns from system calls we mark the page containing the return address as not accessible in the shadow page table. The log is transmitted via network to the trusted host.

5 Experimental Evaluation

We implemented a prototype of the VMM and of the routines for the three analyses described in Section 4. Our current implementation of HyperSleuth is specific for the Microsoft Windows XP (32-bit) operating system. While the core of HyperSleuth is mostly OS-independent, the routines for the analysis (e.g., the enumeration of running processes and of active network connections) are OS-dependent and may require to be slightly adapted to provide support for different operating systems.

In this section we discuss the experimental results concerning the launch of HyperSleuth, the lazy physical memory dumper, and the lie detector. To this

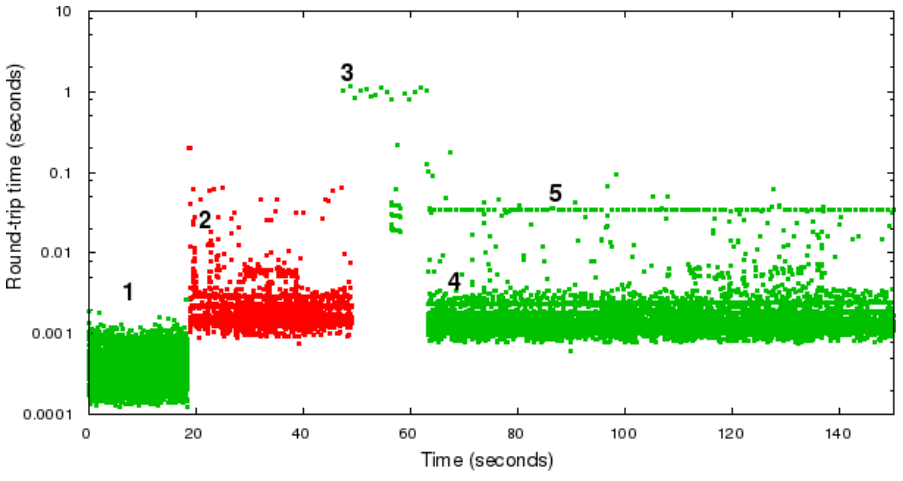


Fig. 5. Round-trip time of the queries performed against the compromised production DNS server before (1) and after (2) the launch of HyperSleuth and (3–5) during the lazy dump of the physical memory (the scale of the ordinate is logarithmic).

end, we simulated the compromised production system using an Intel Core i7, with 3GB RAM, and a Realtek RTL8139 100Mbps network card. Note that we disabled all cores of the CPU but one, since the VMM currently supports a single core. We simulated the trusted host using a laptop. We used the trusted host to attest the correct establishment of the dynamic root of trust and to collect and subsequently analyze the results of the analysis.

5.1 HyperSleuth Launch and Lazy Dump of the Physical Memory

To evaluate the cost of launching HyperSleuth, the base overhead of the VMM, and the cost of the lazy physical memory dumper we simulated the following scenario. A production DNS server was compromised and we used HyperSleuth to dump the entire content of the physical memory when the server was under the heaviest possible load. We used an additional laptop, located on the same network, to flood the DNS server with queries and to measure the instantaneous round-trip time of the queries. About 20 seconds after we started the flood, we launched HyperSleuth; 25 seconds later we started to dump the content of the memory.

Figure 5 summarizes the results of our experiments. The graph shows the round-trip time of the queries sent to the compromised DNS server over time. For the duration of the experiment, the compromised machine was able to handle all the incoming DNS queries, and no query timed out. Before launching HyperSleuth the average round-trip time was $\sim 0.34ms$ (mark 1 in Figure 5). Just after the launch, we observed an initial increase of the round-trip time to about $0.19s$ (mark 2 in Figure 5). This increase was caused by the bootstrap of the dynamic root of trust and then by the launch of the VMM, which must

be performed atomically. After the launch, the round-trip time quickly stabilized around $1.6ms$, less than five times the round-trip time without the VMM. The overhead introduced by the VMM was mostly caused by the handling of the shadow page table. When we started the dump of the physical memory we observed another and steeper peak (mark 3 in Figure 5). We were expecting this behavior since there are a lot of writable memory pages that are frequently accessed (e.g., the stack of the kernel and of the user-space processes and the global variables of the kernel) and that, most likely, are written each time the corresponding process is scheduled. Thus, the peak was caused by the massive number of write accesses to pages not yet dumped. A dozen of seconds later the round-trip time stabilized again around $1.6ms$ (mark 4 in Figure 5). That corresponds to the round-trip time observed before we started the dump. Indeed, the most frequently written pages were written immediately after the dump was started, and the cost of the dump of a single page was much less than the round-trip time and was thus unnoticeable. The regular peaks around $32ms$ about every second (mark 5 in Figure 5) were instead caused by the periodic dump of non-written pages. Since the system was under heavy load, it never entered an idle loop. Thus, the dump was forced after every second of uninterrupted CPU activity. More precisely, the dumper was configured to dump 64 physical pages about every second. Clearly, the number of non-written pages to be dumped when either the system enters the idle loop, or the duration of uninterrupted CPU activity hits a certain threshold, is a parameter that can be tuned accordingly to the urgency of the analysis, to how critical the system is, and to the throughput of the network.

In conclusion, the dump of the whole physical memory of the system (3GB of RAM), in the setting just described, required about 180 minutes and the resulting dump could be analyzed using an off-the-shelf tool, such as Volatility [18]. The total time could be further decreased by increasing the number of physical pages dumped periodically, at the cost of a higher average round-trip time. It should also be pointed out that, on a 1Gbps network, we could increase the number of physical pages dumped every second to 640, without incurring in any additional performance penalty. In this case, the whole physical memory (3GB) would be dumped in just ~ 18 minutes. It is important to remark that although HyperSleuth, and in particular the algorithm for dumping lazily the memory, introduces a non-negligible overhead, we were able to dump the entire content of the memory without interrupting the service (i.e., no DNS query timed out). On the other hand, if the memory were dumped with traditional (atomic) approaches the dump would require, in the ideal case, about 24 seconds, 50 seconds, and 4 minutes respectively on a 1Gbps network, on a 480Mbps FireWire channel, and on a 100Mbps network (these estimations are computed by dividing the maximum throughout of the media by the amount of data to transmit). In these cases, the production system would have not been able to handle any incoming request, for the entire duration of the dump.

Table 1. Results of the evaluation of HyperSleuth's lie detector with seven different malware (all equipped with a root-kit component)

Sample	Characteristics	Detected?
FU	DKOM	✓
FUTo	DKOM	✓
HaxDoor	DKOM, SSDT hooking, API hooking	✓
HE4Hook	SSDT hooking	✓
NtIllusion	DLL injection	✓
NucleRoot	API hooking	✓
Sinowal	MBR infection, Run-time patching	✓

5.2 Lie Detection

Table 1 summarizes the results of the experiments we performed to assess the efficacy of the lie detection module. To this end, we used seven malware samples, each of which included a root-kit component to hide the malicious activity performed on the infected system. We used HyperSleuth's lie detector to detect the hidden activities. The results testify that our approach can be used to detect both user- and kernel-level root-kits.

For each malware sample we proceeded as follows. First, we let the malware infect the untrusted system. Then, we launched HyperSleuth on the compromised host and triggered the execution of the lie detector. The module performed the analysis, first by leveraging the in-guest utility, and then by collecting the same information directly from the VMM through OS-aware inspection. The results were sent separately to the trusted host. On the trusted host we compared the two views of the state of the system and, in all cases, we detected some discrepancies between the two. These discrepancies were all caused by lies. That is, the state visible to the in-guest utility was altered by the root-kit, while the state visible to HyperSleuth VMM was not.

As an example, consider the FUTo root-kit. This sample leverages direct kernel object manipulation (DKOM) techniques to hide certain kernel objects created by the malware (e.g., processes) [1]. Our current implementation of the lie detector counteracts DKOM through a series of analyses similar to those implemented in RAIDE [20]. Briefly, those analyses consist in scanning some internal structures of the Windows kernel that the malware must leave intact in order to preserve its functionalities. Thus, when we compared the trusted with the untrusted view of the state of the system we noticed a process that was not present in the untrusted view produced by the in-guest utility. Another interesting example is NucleRoot, a root-kit that hooks Windows' System Service Descriptor Table (SSDT) to intercept the execution of several system calls and to filter out their results, in order to hide certain files, processes, and registry keys. In this case, by comparing the two views of the state of the system, we observed that some registry keys related to the malware were missing in the untrusted view. Although we have not yet any empirical proof, we speculate the even rootkits like Shadow Walker [2] would be detected by our lie detector since our approach

allows to inspect the memory directly, bypassing a malicious page-fault handler and bogus TLBs' entries.

6 Discussion

We presented HyperSleuth from a technical prospective. The decisions we made in designing and implementing HyperSleuth were mostly motivated by the intent of minimizing the dependencies on the hardware and of maximizing the portability. Therefore, we always opted for pure software-based approaches (e.g., to secure the launch of the VMM and to virtualize the MMU), whenever possible. However, since HyperSleuth is a framework for performing live forensic analyses, it is important to reason about its probatory value. From such a prospective, we must take into account that the trustworthiness of the results of the analyses depends on the trust people have in the tool that generated the results. To strengthen its probatory value, all HyperSleuth's components should be verified in order to prove that their code meets all the expectations [21]. At this aim, in the future we plan to further decrease the size of HyperSleuth's code base in order to ease its verifiability (e.g., by leveraging hardware-based attestation solutions, such as the TPM).

HyperSleuth's effectiveness depends on the impossibility to detect its presence from the guest. Although the VMM is completely isolated from the guest, the malware might attempt to detect HyperSleuth by trying to install another VMM. One approach to contrast such attempts is to let the malware believe that virtualization support is not available at all.

7 Related Work

The idea of leveraging a virtual machine monitor to perform sophisticated runtime analyses, with the guarantee that the results cannot be tampered by a malicious attacker, has already been widely explored in the literature. Garfinkel *et al.* were the first to propose to use a VMM to perform OS-aware introspection [6], and subsequently the idea was further elaborated [22, 5]. Other researchers instead proposed to use a VMM to protect the guest OS from attacks by supervising its execution, both with a software-based VMM [8] and by leveraging hardware support for virtualization [9]. Similar ideas were also suggested by other authors [7, 23]. In [24] Chen *et al.* proposed a solution to protect applications' data even in the presence of a compromised operating system. More recently, Vasudevan *et al.* proposed XTREC, a lightweight framework to record securely the execution control flow of all code running in an untrusted system [25]. Unfortunately, in order to guarantee that the analyses they perform cannot be tampered by an attacker, all the aforementioned solutions must take control of the system before the guest is booted, and cannot be removed until the guest is shut down. On the contrary, HyperSleuth can be installed as the compromised system runs, and, when the analyses are completed, it can be removed on-the-fly. The idea to take advantage of the possibility to install a VMM on a running system was also

sketched in [26], and later investigated in our previous research work to realize HyperDbg, a transparent kernel-level debugger [27].

Several researchers proposed to use VMMs to implement malware that are particularly hard to detect and to eradicate. SubVirt was one of the first prototypes that employed this technique [28]. However, being implemented using a software-based VMM, the installation of Subvirt required to reboot the machine, and the malware also introduced a noticeable run-time overhead in the infected target. Later, the Blue Pill malware started to exploit the hardware-assisted supports for virtualization to implement an efficient VMM-based malware that is able to infect a machine as it runs, without the need for reboot [10]. HyperSleuth was inspired by this malware.

8 Conclusion

We presented HyperSleuth, a framework for constructing forensic tools that leverages the virtualization extensions provided by commodity hardware to guarantee that the results of the analyses cannot be altered, even by an attacker with kernel-level privileges. HyperSleuth consists in a tiny hypervisor that is installed on a potentially compromised system as it runs, and a secure loader that installs the hypervisor and verifies its integrity. We developed a proof-of-concept prototype of HyperSleuth and, on top of it, we implemented three forensic analysis applications: a lazy physical memory dumper, a lie detector, and a system call tracer. Our experimental evaluation testified the effectiveness of the proposed approach.

References

1. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional, Reading (2005)
2. Sparks, S., Butler, J.: Shadow Walker. Raising The Bar For Windows Rootkit Detection. Phrack Magazine 11(63) (2005)
3. AMD, Inc.: AMD Virtualization, `www.amd.com/virtualization`
4. Intel Corporation: Intel Virtualization Technology, `http://www.intel.com/technology/virtualization/`
5. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security (2008)
6. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proceedings of the Network and Distributed Systems Security Symposium. The Internet Society, San Diego (2003)
7. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: An Architecture for Secure Active Monitoring Using Virtualization. In: Proceedings of the IEEE Symposium on Security and Privacy (2008)
8. Riley, R., Jiang, X., Xu, D.: Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In: Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (2008)

9. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In: Proccedings of the ACM Symposium on Operating Systems Principles. ACM, New York (2007)
10. Rutkowska, J.: Subverting Vista Kernel For Fun And Profit. Black Hat USA (2006)
11. McCune, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for tcb minimization. In: Proceedings of the ACM European Conference in Computer Systems (2008)
12. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In: Proceedings of ACM Symposium on Operating Systems Principles (2005)
13. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: Swatt: Software-based attestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy (2004)
14. Martignoni, L., Paleari, R., Bruschi, D.: Conqueror: tamper-proof code execution on legacy systems. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment. LNCS. Springer, Heidelberg (2010)
15. Grawrock, D.: Dynamics of a Trusted Platform: A Building Block Approach. Intel Press, Hillsboro (2009)
16. Carbone, M., Zamboni, D., Lee, W.: Taming virtualization. IEEE Security and Privacy 6(1) (2008)
17. Smith, J.E., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, San Francisco (2005)
18. Volatile Systems LLC: Volatility, `http://www.volatilesystems.com/`
19. Forrest, S., Hofmeyr, S.R., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: Proceedings of the IEEE Symposium on Security and Privacy (1996)
20. Butler, J., Silberman, P.: RAIDE: Rookit analysis identification elimination. In: Black Hat USA (2006)
21. Franklin, J., Seshadri, A., Qu, N., Datta, A., Chaki, S.: Attacking, Repairing, and Verifying SecVisor: A Retrospective on the Security of a Hypervisor. Technical Report, Carnegie Mellon University (2008)
22. Jiang, X., Wang, X.: "out-of-the-box" monitoring of VM-based high-interaction honeypots. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (2007)
23. Sharif, M., Lee, W., Cui, W., Lanzi, A.: Secure In-VM Monitoring Using Hardware Virtualization. In: Proceedings of the ACM Conference on Computer and Communications Security (2009)
24. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. Operating Systems Review 42(2) (2008)
25. Perrig, A., Gligor, V., Vasudevan, A.: XTREC: secure real-time execution trace recording and analysis on commodity platforms. Technical Report, Carnegie Mellon University (2010)
26. Sahita, R., Warrier, U., Dewan, P.: Dynamic software application protection. Technical Report, Intel Corporation (2009)
27. Fattori, A., Paleari, R., Martignoni, L., Monga, M.: HyperDbg: a fully transparent kernel-level debugger, `http://code.google.com/p/hyperdbg/`
28. King, S.T., Chen, P.M., Wang, Y.M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing malware with virtual machines. In: Proceedings of IEEE Symposium on Security and Privacy (2006)

Hybrid Analysis and Control of Malware

Kevin A. Roundy and Barton P. Miller

Computer Sciences Department
University of Wisconsin
{roundy,bart}@cs.wisc.edu

Abstract. Malware attacks necessitate extensive forensic analysis efforts that are manual-labor intensive because of the analysis-resistance techniques that malware authors employ. The most prevalent of these techniques are code unpacking, code overwriting, and control transfer obfuscations. We simplify the analyst's task by analyzing the code prior to its execution and by providing the ability to selectively monitor its execution. We achieve pre-execution analysis by combining static and dynamic techniques to construct control- and data-flow analyses. These analyses form the interface by which the analyst instruments the code. This interface simplifies the instrumentation task, allowing us to reduce the number of instrumented program locations by a hundred-fold relative to existing instrumentation-based methods of identifying unpacked code. We implement our techniques in SD-Dyninst and apply them to a large corpus of malware, performing analysis tasks such as code coverage tests and call-stack traversals that are greatly simplified by hybrid analysis.

Keywords: malware analysis, forensics, hybrid, de-obfuscation, packed code, self-modifying code, obfuscated code.

1 Introduction

Malicious software infects computer systems at an alarming rate, causing economic damages that are estimated at more than ten billion dollars per year [1]. Immediately upon discovering a new threat, analysts begin studying its code to determine damage done and information extracted, and ways to curtail its impact; analysts also study the malware so they can recover infected systems and construct defenses. Thus, a primary goal of malware authors is to make these tasks as difficult and resource intensive as possible. This explains why 90% of malware binaries employ *analysis-resistance* techniques [9], the most prevalent of which are the run-time unpacking of compressed and encrypted code, run-time modifications to existing code, and obfuscations of control transfers in the code. Security companies detect thousands of new malware samples each day [45], yet despite the importance and scale of this problem, analysts continue to resort to manual-labor intensive methods.

Analysts accomplish their tasks by studying the malware's overall structure to identify the relevant code and then analyzing it thoroughly. Unfortunately, analysis-resistance techniques force the analyst out of the usual mode of binary

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 317–338, 2010.

analysis, which involves statically analyzing the binary prior to instrumenting regions of interest and performing a controlled execution of the program. Instead, the analyst must execute the malicious code to analyze it, as static analysis can fail to analyze dynamically generated, modified, and obfuscated code. The analyst must therefore construct a virtual environment that allows the malware to interact with other hosts and that is sufficiently convincing that the malware will display its normal behavior while executing in isolation from the outside world. Many analysts prefer the analyze-then-execute model and therefore resort to expending considerable manual effort to strip analysis-resistance features from malicious binaries [12,40].

The goal of our research is to simplify malware analysis by enabling a return to the traditional analyze-then-execute model, which has the benefit of bringing the malicious code under the analyst's control before it executes. We address these goals by combining static and dynamic techniques to construct and maintain the control- and data-flow analyses that form the interface through which the analyst understands and instruments the code. A key feature of our approach is its ability to update these analyses to include dynamically unpacked and modified code before it executes. Our work makes the following contributions:

- Pre-execution analysis and instrumentation makes it possible for the analyst to control the execution of malicious code. For example, our work allows interactions with other infected hosts to be simulated through instrumentation's ability to patch the program, removing the need for complex virtual environments involving multiple hosts. Additionally, our work can complement a virtualization strategy by identifying and disabling the malware's attempts to detect its virtual-machine environment [36].
- We give the analyst the ability to instrument malware intuitively and efficiently by providing data-flow analysis capabilities and a control flow graph (CFG) as an interface to the code. For example, the CFG allows us to find transitions to dynamically unpacked code by instrumenting the program's statically unresolved control transfers (see Section 5). By contrast, prior instrumentation-based unpacking tools did not maintain a CFG and therefore had to monitor all of the program's control transfers and memory writes to detect the execution of code that is written at run-time [23,41]. We achieve a hundred-fold reduction in the number of monitored program locations.
- Our structural analysis allows analysts to be selective in the components they monitor, the operations in those components that they select, and in the granularity of data they collect. Current tools that can monitor analysis-resistant malware do not provide flexible instrumentation mechanisms; they trace the program's execution at a uniform granularity, either providing fine-grained traces at the instruction or basic-block level [17,36], or coarse grained traces (e.g., at interactions with the OS) [52]. These tools either bog the analyst down with irrelevant information (a significant problem for inexperienced analysts [42]), or can only give a sketch of the program's behavior.
- By combining static and dynamic techniques we allow the analyst to find and analyze code that is beyond the reach of either static or dynamic analysis

alone, thereby providing a fuller understanding of the malware's possible behavior. Prior combinations of static and dynamic analysis only operate on non-defensive code, and only find and disassemble the code [34] or produce their analysis results only after the program has fully executed [28].

Analysts have controlled and monitored malicious code either by executing the malware in a process that they control through the debugger interface [44], or by executing the malware in a virtual machine [36]. There are advantages to both approaches. The debugger approach makes it easy to collect process information and intercept events, and allows for the creation of lightweight tools that do not have to worry about anti-emulation techniques [36]. Among the benefits of virtual machines are that they isolate the underlying system from the malware's effects, they provide the ability to revert the machine to a clean state or to a decision point, and allow for stealthy monitoring of the program's execution [17,36]. While in this paper we demonstrate an instrumentation and analysis tool that executes malicious processes through the debugger interface, our techniques are orthogonal to this choice and benefit both scenarios. For example, in the former case, pre-execution analysis and control allows the analyst to limit the damage done to the system, while the latter case benefits from the ability to detect and disable anti-emulation techniques.

Our analysis and instrumentation tool is not the first to analyze code prior to its execution (e.g., Dyninst [22], Vulcan [48]), but existing tools rely exclusively on static analysis, which can produce incomplete information even for binaries that are generated by standard compilers. Despite recent advances in identifying functions in stripped compiler-generated binaries, on the average, 10% of the functions generated by some compilers cannot be recognized by current techniques [43], and even costly dataflow analyses such as pointer aliasing may be insufficient to predict the targets of pointer-based control transfers [5,21].

Most malware binaries make analysis and control harder by employing the analysis-resistance techniques of code packing, code overwriting, and control transfer obfuscations. *Code packing* techniques, wherein all or part of the binary's malicious code is compressed (or encrypted) and packaged with code that decompresses the malicious payload into the program's address space at run-time, are present in 75% of all malware binaries [8,50]. Dealing with dynamic code unpacking is complicated by programs that unpack code in stages, by the application of multiple code-packing tools to a single malicious binary, and by a recent trend away from well-known packing tools, so that most new packed binaries use custom packing techniques [9,10].

To further complicate matters, many packing tools and malicious programs *overwrite* code at run-time. While code unpacking impacts static analysis by making it incomplete, code overwriting makes the analysis invalid *and* incomplete. A static analysis may yield little information on a self-modifying program, as potentially little of the code is exposed to analysis at any point in time [2].

Malware often uses *control-transfer obfuscations* to cause static analysis algorithms to miss and incorrectly analyze large swaths of binary code. In addition to the heavy use of indirect control transfers, obfuscated binaries commonly include

non-conventional control transfer sequences (such as the use of the return instruction as an indirect jump), and signal- and exception-based control transfers [18]. Additionally, malicious binaries and packers often contain hand-written assembly code that by its nature contains more variability than compiler-generated code, causing problems for most existing code-identification strategies, as they depend on the presence of compiler-generated instruction patterns [24,43].

We analyze binaries by first building a CFG of the binary through static parsing techniques. As the binary executes, we rely on dynamic instrumentation and analysis techniques to discover code that is dynamically generated, hidden by obfuscations, and dynamically modified. We then re-invoke our parsing techniques to update our CFG of the program, identifying the new code and presenting the updated CFG to the analyst. The structural information provided by our analysis allows us to discover new code by instrumenting the program lightly, only at control transfer instructions whose targets cannot be resolved through static analysis, making our tool's execution time comparable to that of existing unpacking tools despite the additional cost that we incur to provide an updated CFG. (see Section 8).

Other analysis resistance techniques that can be employed by malicious program binaries include anti-debugging checks, anti-emulation checks, timing checks, and anti-tampering (e.g., self-checksumming) techniques. Since our current implementation uses the debugger interface, we have neutralized the common anti-debugging checks [18]. In practice, anti-debugging and timing checks leave footprints that are evident in our pre-execution analysis of the code and that the analyst can disable with the instrumentation capabilities that we provide. Our research is investigating multiple alternatives for neutralizing the effect of anti-tampering techniques, but this work is beyond the scope of this paper.

We proceed by discussing related work in Section 2 and we give an overview of our techniques and algorithm in Section 3. Our code-discovery techniques are described in Sections 4-7. In Section 8 we show the utility of our approach by applying our tool to analysis-resistant malware, and to synthetic samples produced by the most prevalent binary packing tools. We conclude in section 9.

2 Related Work

Our work is rooted in the research areas of program binary analysis, instrumentation, and unpacking.

Analysis. Static parsing techniques can accurately identify 90% or more of the functions in compiler-generated binaries despite the lack of symbol information [43], but are much worse at analyzing arbitrarily obfuscated code [24,51], and cannot analyze the packed code that exists in most malicious binaries [9]. Thus, most malware analysis is dynamic and begins by obtaining a trace of the program's executed instructions through single-step execution [17], dynamic instrumentation [41], or the instrumentation capabilities of a whole-system emulator [32]. The resulting trace is used to construct an analysis artifact such as a visualization of the program's unpacking behavior [42], a report of its operating

system interactions [6], or a representation that captures the evolving CFG of a self-modifying program [4]. As these analysis artifacts are all produced after monitoring the program's execution, they are potential clients of our analysis-guided instrumentation techniques rather than competitors to them.

Madou et al. [28] and Cifuentes and Emmerik [13] combine static and dynamic techniques to identify more code than is possible through either technique alone. Madou et al. start from an execution trace and use control-flow traversal parsing techniques to find additional code, whereas Cifuentes and Emmerik start with speculative static parsing and use an instruction trace to reject incorrect parse information. Their hybrid approaches to CFG-building are similar to ours in spirit, but they analyze only code that is statically present in the binary as they lack the ability to capture dynamically unpacked and overwritten code.

Instrumentation. Existing tools that provide analysis-guided binary instrumentation [22,49,48] cannot instrument code that is obfuscated, packed, or self-modifying, as their code identification relies exclusively on static analysis. Our tool uses Dyninst's [22] dynamic instrumentation and analysis capabilities, updating its analysis of the code through both static and dynamic code-capture techniques, prior to the code's execution.

The BIRD dynamic instrumenter [34] identifies binary code by augmenting its static parse with a run-time analysis that finds code by instrumenting control transfers that could lead to unknown code areas. BIRD works well on compiler-generated programs, but does not handle self-modifying programs and performs poorly on programs that are packed or obfuscated, as it is not optimized for extensive dynamic code discovery (it uses trap instructions to instrument all return instructions and other forms of short indirect control transfers that it discovers at runtime). BIRD also lacks a general-purpose interface for instrumentation and does not produce analysis tools for the code it identifies.

Other dynamic instrumentation tools forgo static analysis altogether, instead discovering code as the program executes and providing an instruction-level interface to the code (e.g., PIN [27], Valgrind [7]). These tools can instrument dynamically unpacked and self-modifying code, but do not defend against analysis-resistance techniques [18]. As with BIRD, the lack of a structural analysis means that it is difficult to selectively instrument the code and that it may not be possible to perform simple-sounding tasks like hooking a function's return values because compiler optimizations (and obfuscations) introduce complexities like shared code, frameless functions, and tail calls in place of return statements.

Unpacking. The prevalence of code packing techniques in malware has driven the creation of both static and dynamic approaches for detecting packed malicious code. Some anti-virus tools (e.g., BitDefender [8]) create static unpackers for individual packer tools at significant expense, but this approach will not scale with the explosive growth rate of new packing approaches [9,35]. Anti-virus tools also employ "X-Ray" techniques that can statically extract the packed contents of binaries that employ known compression schemes or weak encryption [37]. Coogan et al. [15] use static analysis to extract the unpacking routine from a

packed binary and then use the extracted routine to unpack it. These static approaches are unsuccessful when confronted with malware that employs multiple packing layers (e.g., Rustock [12]), and Coogan et al.'s static techniques are also unable to deal with heavily obfuscated code [33,46].

Most dynamic unpacking tools take the approach of detecting memory locations that are written to at run-time and later executed as code. OmniUnpack [30], Saffron [41], and Justin [20] approach the problem at a memory-page granularity by modifying the operating system to manipulate page write and execute permissions so that both a write to a page and a subsequent execution from that page result in an exception that the tool can intercept. This approach is efficient enough that it can be used in an anti-virus tool [30], but it does not identify unpacked code with much precision because of its memory-page granularity.

Other unpackers identify written-then-executed code at a byte level by tracing the program's execution at a fine granularity and monitoring all memory writes. EtherUnpack [17] and PolyUnpack [44] employ single-step execution of the binary, whereas Renovo [23] and "Saffron for Intel-PIN" [41] use the respective instruction-level instrumentation capabilities of the Qemu whole-system emulator [7] and the PIN instrumenter [27]. By contrast, our analysis-guided instrumentation allows us to unpack *and analyze* program binaries with a hundred-fold reduction in instrumented program locations and comparable execution times.

3 Technical Overview

Our hybrid algorithm combines the strengths of static and dynamic analysis. We use static parsing techniques to analyze code before it executes, and dynamic techniques to capture packed, obfuscated, and self-modifying code. Hybrid analysis allows us to provide *analysis-guided dynamic instrumentation* on analysis-resistant program binaries for the first time, based on the following techniques:

Parsing. Parsing allows us to find and analyze binary code by traversing statically analyzable control flow starting from known entry points into the code. No existing algorithm for binary code analysis achieves high accuracy on arbitrarily obfuscated binaries, so we create a modified control-flow traversal algorithm [47] with a low false-positive rate. Our initial analysis of the code may be incomplete, but we can fall back on our dynamic capture techniques to find new entry points into the code and use them to re-seed our parsing algorithm.

Dynamic Capture. Dynamic capture techniques allow us to find and analyze code that is missed by static analysis either because it is not generated until run-time or because it is not reachable through statically analyzable control flow. Our static analysis of the program's control flow identifies control transfer instructions that may lead to un-analyzed code; we monitor these control transfers using dynamic instrumentation, thereby detecting any transition to un-analyzed

```
1. Load the program into memory, paused at its entry point
2. Remove debugging artifacts
3. Parse from known entry points
4. Instrument newly discovered code
5. Resume execution of the program
6. Handle code discovery event, adding new entry points
7. Goto 3
```

Fig. 1. Algorithm for binary code discovery, analysis, and instrumentation

code in time to analyze and instrument it before it executes. This approach is similar to BIRD's [34], but monitors a smaller set of control transfers.

Code Overwrite Monitoring. Code overwrites invalidate portions of an existing code analysis and introduce new code that has not yet been analyzed. We adapt DIOTA's [29] mechanism for detecting code overwrites by write-protecting memory pages that contain code and handling the signals that result from write attempts. Accurately detecting when overwriting ends is important, as it allows us to update our analysis only once when large code regions are overwritten in small increments. We detect the end of code overwriting in a novel way by using our structural analysis of the overwrite code to detect any loops that enclose the write operations, allowing us to delay the analysis update until the loop exits.

Signal- and Exception-Handler Analysis. We use dynamic analysis to resolve signal- and exception-based control transfer obfuscations [18,38]. We detect signal- and exception-raising instructions and find their dynamically registered handlers through standard techniques, and then add the handlers to our analysis and instrument them to control their execution.

Figure 1 illustrates how we combine the above techniques into an iterative algorithm that allows us to provide analysis-guided dynamic instrumentation of analysis-resistant program binaries. The key feature of this algorithm is that it allows all of the program's code to be analyzed and instrumented before it executes. Our algorithm's incremental instrumentation of the code is similar to Mirgorodskiy and Miller's use of "self-propelled instrumentation" to trace a program's execution [31], but we also analyze and instrument analysis-resistant code, whereas they can instrument only statically analyzable code.

4 Parsing

The purpose of our parsing algorithm is to accurately identify binary code and analyze the program's structure, producing an interprocedural control flow graph of the program. Existing parsing techniques for arbitrarily obfuscated code have attempted to identify code with good accuracy and coverage, and have come up short on both counts [24]. Instead, we prioritize accurate code identification, as an incorrect parse can cause incorrect program behavior by leading to the instrumentation of non-code bytes, and is ultimately not very useful. The competing

goal of good coverage is relatively less important, because our dynamic techniques compensate for lapses in coverage by capturing statically un-analyzable code at run-time and triggering additional parsing.

Control-flow traversal parsing [47] is the basis for most accurate parsing techniques, but it makes three unsafe assumptions about control flow that can reduce its accuracy. First, it assumes that function-call sites are always followed by valid code sequences. Compilers violate this assumption when generating calls to functions that they know to be non-returning, while obfuscated programs (e.g., Storm Worm [39]) often contain functions that return to unexpected locations by tampering with the call stack [25]. Second, the algorithm assumes that control flow is only redirected by control transfer instructions. Obfuscated programs often use an apparently normal instruction to raise a signal or exception, thereby transferring control to code that is hidden in a signal or exception handler [18]. The handler can further obfuscate control flow by telling the operating system to resume execution away from the signal- or exception-raising instruction, potentially causing non-code bytes to be parsed following the instruction [38]. Third, the algorithm assumes that both targets of conditional branch instructions can be taken and therefore contain valid code. Program obfuscators can exploit this assumption by creating branches with targets that are never taken, thereby diluting the analysis with junk code that never executes [14].

In our experience with analysis-resistant binaries, we have found that by far the most targeted vulnerability is the assumption that code follows each call instruction, and we have addressed this vulnerability in our current parsing algorithm. We detect and resolve signal- and exception-based obfuscations at run-time (see Section 7), when we analyze and instrument any hidden code and correct our analysis to include the obfuscated control transfer. The use of branches with targets that are never taken dilutes the analysis with junk code but has thus far not been a problem for our hybrid analysis and instrumentation techniques. Our ongoing work will improve upon our parser's accuracy by adding static detection of some fault-based control transfers and never-taken branch targets, thereby making our instrumentation of the program safer and more efficient. In the meantime, our current parsing algorithm achieves significant accuracy improvements relative to existing techniques for parsing obfuscated code, allowing us to analyze and instrument most analysis-resistant programs.

Non-returning Calls. When a called function either never returns or returns to an unexpected location by tampering with the call stack [25], one or more junk bytes may follow the function call site. The simplest approach to this problem would be to adopt the assumption made by BIRD [34] and Kruegel et al.'s obfuscated code parser [24], that function calls never return, and then rely on run-time monitoring of return instructions to discover code that follows call sites. This runtime-discovery approach is taken by BIRD, and while it is our technique of last resort, our data-flow analysis of called functions can often tell us whether the function will return, and to where, thereby increasing the code coverage attained through parsing and avoiding unnecessary instrumentation.

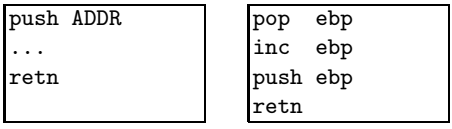


Fig. 2. Code sequences that tamper with return addresses on the call stack

We take advantage of the depth-first nature of our parsing algorithm to use the analysis of called functions in deciding whether or not to continue parsing after call instructions. We do not resume parsing after the call site if our analysis of the called function contains no return instructions, or if our static call stack analysis [26] of the function detects that it tampers with the stack. Our call-stack analysis emulates the function's stack operations to detect whether the function alters its return address, either by overwriting the address or by imbalancing the call stack. Figure 2 illustrates two call-stack tampering tricks used by the ASPack packer that are easily detected by our analysis. Figure 2a shows an instruction sequence that transfers control to `ADDR` upon executing the return instruction, while Figure 2b shows a sequence that increments the return address of a function by a single byte. In both cases, our call-stack analysis informs the parser of the actual return address of the called function, and the byte immediately following the call site is not parsed as code.

5 Dynamic Capture

Having found and analyzed as much of the code as possible by traversing the program's statically analyzable control flow, we turn to *dynamic capture* techniques to find code that is not statically analyzable. Statically un-analyzable code includes code that is present in the binary but is reachable only through pointer-based address calculations, and code that is not initially present because it is dynamically unpacked. Our approach to both problems lies in monitoring those control transfers whose targets are either unknown or invalid when we originally parse them. More precisely, we use dynamic capture instrumentation to monitor the execution of instructions that meet one of the following criteria:

– *Control transfer instructions that use registers or memory values to determine their targets.* Obfuscated programs often used indirect jump and call instructions to hide code from static analysis. For example, the FSG packer has an indirect function call for every 16 bytes of bootstrap code. We determine whether indirect control transfers leave analyzed code by resolving their targets at run-time with dynamic instrumentation. In the case of indirect call instructions, when our parser cannot determine the call's target address, it also cannot know if the call will return, so it conservatively assumes that it does not; our instrumentation allows us to trigger parsing both at call target and after the call site, if we can determine that the called function returns.

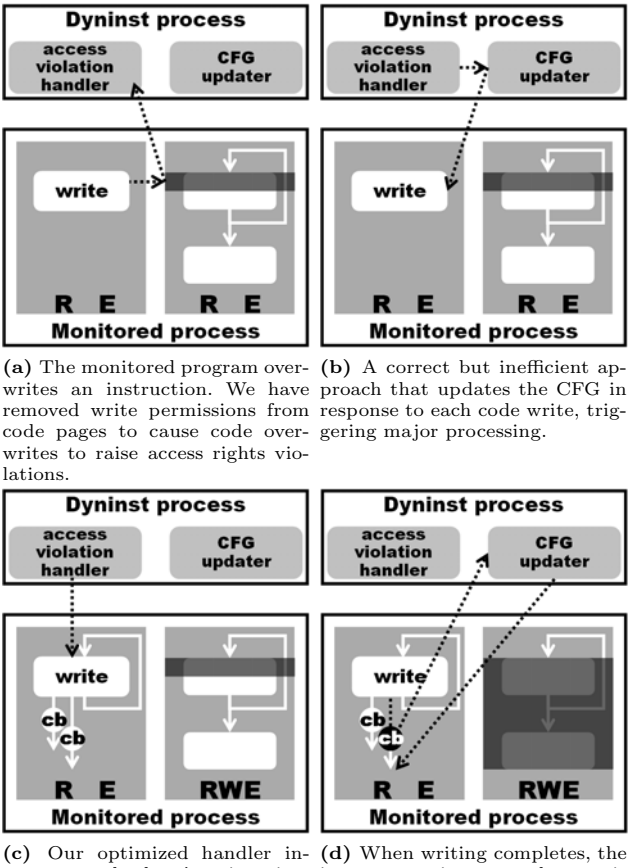
- *Return instructions of possibly non-returning functions.* Return instructions are designed to transfer execution from a called function to the caller at the instruction immediately following the call site; unfortunately they can be misused by tampering with the call stack. As detailed in Section 4, during parsing we attempt to determine whether called functions return normally so that we can continue parsing after call sites to those functions. If our analysis is inconclusive we instrument the function's return instructions.
- *Control transfer instructions into invalid or uninitialized memory regions.* Control transfers to dynamically unpacked code can appear this way, as code is often unpacked into uninitialized (e.g., UPX) or dynamically allocated memory regions (e.g., NSPack). Our instrumentation of these control transfer instructions executes immediately prior to the transfer into the region, when it must contain valid code, allowing us to analyze it before it executes.
- *Instructions that terminate a code sequence by reaching the end of initialized memory.* Some packer tools (e.g., UPack) and custom-packed malware (e.g., Rustock [12]) transition to dynamically unpacked code without executing a control transfer instruction. These programs map code into a larger memory region so that a code sequence runs to the end of initialized memory without a terminating control transfer instruction. The program then unrolls the remainder of the sequence into the region's uninitialized memory so that when it is invoked, control flow falls through into the unpacked code. We trigger analysis of the unpacked instructions by instrumenting the last instruction of any code sequence that ends without a final control transfer instruction.

Our dynamic capture instrumentation supplies our parser with entry points into un-analyzed code. Before extending our analysis by parsing from these new entry points, we determine whether the entry points represent un-analyzed functions or if they are extensions to the body of previously analyzed functions. We treat call targets as new functions, and treat branch targets as extensions to existing functions (unless the branch instruction and its target are in different memory regions). The target of a non-obfuscated return instruction is always immediately preceded by an analyzed call instruction, in which case we parse the return instruction's target as an extension of the calling function. When a return target is not immediately preceded by a call instruction, we conclude that the call stack has been tampered with and parse the target as a new function.

Cost issues arise from our use of the Dyninst instrumentation library [22] because it monitors programs from a separate process that contains the analysis and instrumentation engine. The problem is that our code-discovery instrumentation requires context switching between the two processes to determine whether monitored control transfers lead to new or analyzed code. We reduce this overhead by caching the targets of these instructions in the address space of the monitored program, and context switching to Dyninst only for cache misses.

6 Response to Overwritten Code

Code overwrites cause significant problems for binary analysis. Most analysis tools cannot analyze overwritten code because they rely on static CFG representations



(a) The monitored program overwrites an instruction. We have removed write permissions from code pages to cause code overwrites to raise access rights violations.

(b) A correct but inefficient approach that updates the CFG in response to each code write, triggering major processing.

(c) Our optimized handler instruments the loop's exit points with callbacks to our CFG updater and restores write permissions to the overwritten page.

(d) When writing completes, the instrumentation at a loop exit triggers a callback to our CFG updater.

Fig. 3. Our approach to detecting code writes is shown in Figure 3a, alternative methods for responding to code writes are shown in Figures 3b and 3c-3d.

of the code. Code overwrites cause problems for CFGs by simultaneously invalidating portions of the CFG and introducing new code that has yet to be analyzed. We have developed techniques to address this problem by updating the program's CFG and analyzing overwritten code before it executes.

To analyze overwritten code before it executes, we can either detect the modified instructions immediately prior to their execution, by checking whether the bytes of each executed instruction have changed [44], or detect writes to code as soon as they occur, by monitoring write operations to analyzed code regions. Monitoring each instruction for changes is expensive because it requires single-step execution of the program. Fortunately, we can efficiently detect write operations that modify code by adapting DIOTA's techniques for intercepting writes to executable code regions [29]. DIOTA monitors writes to all memory

pages that are writable and executable by removing write permissions from those pages, thereby causing writes that might modify code to raise an access-rights violation that DIOTA intercepts. As illustrated in Figure 3a, we have adapted this mechanism for packed binaries, which typically mark most of their memory as writable and executable, by removing write permissions *only* from memory pages that contain *analyzed* code.

A naïve approach based on monitoring writes to code pages might respond to write attempts by emulating each write and immediately updating the analysis, as shown in Figure 3b. Doing so is undesirable for efficiency reasons. Large code regions are often overwritten in small increments, and updating the CFG in response to every code write is unnecessarily expensive, as the analysis does not need to be updated until the overwritten code is about to execute. Instead, we catch the first write to a code page but allow subsequent writes, delaying the update to the window between the end of code overwriting and the beginning of modified code execution. This delayed-update approach divides our response to code overwrites into two components that we now describe in detail: a handler for the access-rights violation resulting from the first write attempt, and a CFG update routine that we trigger before the modified code executes.

6.1 Response to the Initial Access-Rights Violation

When a write to a code page results in an access-rights violation, our first task is to handle the exception. We disambiguate between real access-rights violations and protected-code violations by looking at the write's target address. Protected-code violations are artificially introduced by our code-protection mechanism and we handle them ourselves to hide them from the monitored program. For real access-rights violations we apply the techniques of Section 7 to analyze the registered handler and pass the signal or exception back to the program.

Our handler also decides when to update the CFG, attempting to trigger the update after the program has stopped overwriting its code, but before the modified code executes. A straightforward approach would be to restore write permissions for the overwritten page and remove execute permissions from the page, thereby causing a signal to be raised when the program attempts to execute code from the overwritten page (similar to the approach taken by OmniUnpack [30], Justin [20], and Saffron [41]). Unfortunately, this approach fails in the common case that the write instruction writes repeatedly to its own page, when this approach effectively devolves into single-step execution. Instead, we apply the techniques shown in Figures 3c and 3d to detect the end of overwriting, and delay updating the CFG until then. We perform inter-procedural loop analysis on the execution context of the faulting write instruction to see if the write is contained in a loop, in which case we instrument the loop's exit edges with callbacks to our CFG update routine. We allow subsequent writes to the write-target's code page to proceed unimpeded, by restoring the page's write permissions. When the write loop terminates, one of the loop's instrumented exit edges causes the CFG update routine to be invoked. We take extra precautions if the write loop's code pages intersect with the overwritten pages to ensure that the write loop does not

modify its own code. We safeguard against this case by adding bounds-check instrumentation to all of the loop's write operations so that any modification of the loop's code will immediately trigger our CFG update routine.

Our handler's final task is to save a pre-write copy of the overwritten memory page, so that when writing completes, the CFG update routine can identify the page's modified instructions by comparing the overwritten page to the pre-write copy of the page. If the loop writes to multiple code pages, the first write to each code page results in a separate invocation of our protected-code handler, which triggers the generation of a pre-write copy of the page, and associates it with the write loop by detecting that the write instruction lies within it. Our handler then restores write permissions to the new write-target page and resumes the program's execution. When the write loop finally exits, instrumentation at one of its exit edges triggers a callback to our CFG update routine.

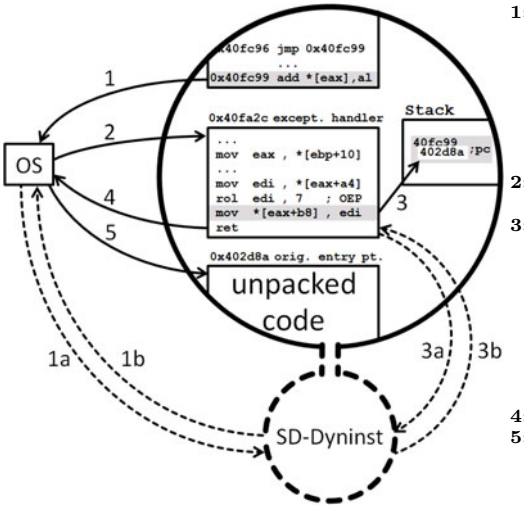
6.2 Updating the Control Flow Graph

We begin updating our analysis by determining the extent to which the code has been overwritten. We identify overwritten bytes by comparing the overwritten code pages with our saved pre-write copies of those pages, and then detemine which of the overwritten bytes belong to analyzed instructions. If code was overwritten, we clean up the CFG by purging it of overwritten basic blocks and of blocks that are only reachable from overwritten blocks. We analyze the modified code by seeding our parser with entry points into the modified code regions. We then inform the analyst of the changes to the program's CFG so that the new and modified functions can be instrumented. After adding our own dynamic capture instrumentation to the new code, we again remove write permissions from the overwritten pages and resume the monitored program's execution.

7 Signal- and Exception-Handler Analysis

Analysis-resistant programs are often obfuscated by signal- and exception-based control flow. Static analyses cannot reliably determine which instructions will raise signals or exceptions, and have difficulty finding signal and exception handlers, as they are usually registered (and often unpacked) at run-time. Current dynamic instrumentation tools do not analyze signal and exception handlers [27,29], whereas we analyze them and provide analysis-based instrumentation on them. This ability to analyze and control the handlers is important on analysis-resistant binaries because the handlers may perform tasks that are unrelated to signal and exception handling (e.g., PECompact overwrites existing code).

Signal and exception handlers can further obfuscate the program by redirecting control flow [38]. When a signal or exception is raised, the operating system gives the handler context information about the fault, including the program counter value. The handler can modify this saved PC value to cause the OS to resume the program's execution at a different address. As shown in step 3 of Figure 4, this technique is used by the "Yoda's Protector" packer to obfuscate



1: A store to address 0 causes an access violation and the OS saves the fault's PC on the call stack.
- ***1a:*** *The OS informs the attached SD-Dyninst process of the exception.*
- ***1b:*** *SD-Dyninst analyzes the registered exception handler, instruments its exit points, and returns to the OS.*

2: The OS calls the program's exception handler.

3: The handler overwrites the saved PC with the address of the program's original entry point.
- ***3a:*** *Instrumentation at the handler's exit point invokes SD-Dyninst.*
- ***3b:*** *SD-Dyninst detects the modified PC value, analyzes the code at that address, and resumes the handler's execution.*

4: The handler returns to the OS.

5: The OS resumes the program's execution at the modified PC value, which is the program's original entry point.

Fig. 4. The normal behavior of an exception-based control transfer used by Yoda's Protector is illustrated in steps 1-5. Steps 1a-1b and 3a-3b illustrate SD-Dyninst's analysis of the control transfer through its attached debugger process.

its control transfer to the program's original entry point (OEP) [16]. Yoda's Protector raises an exception, causing the OS to invoke Yoda's exception handler. The handler overwrites the saved PC value with the address of the program's OEP, causing the OS to resume the program's execution at its OEP.

Analyzing Signal- and Exception-Based Control Flow. We find and analyze handlers by intercepting signals and exceptions at run-time. Signal and exception interception is possible whether we observe the malware's execution from a debugger process or virtual machine monitor (VMM). In our current implementation, SD-Dyninst is apprised of raised signals and exceptions through standard use of the debugger interface. A VMM-based implementation would automatically be apprised of signals and exceptions, and would use VM-introspection techniques to detect which of them originate from malicious processes [19].

As shown in Figure 4, upon notification of the signal or exception, we analyze and instrument the program's registered handlers. We find handlers in Windows programs by traversing the linked list of structured exception handlers that is on the call stack of the faulting thread. Finding handlers is even easier in Unix-based systems because only one handler can be registered to each signal type. We analyze the handler as we would any other function, and mark the faulting instruction as an invocation of the handler.

We guard against the possibility that the handler will redirect control flow by instrumenting it at its exit points. After analyzing the handler, but before it executes, we insert our exit-point instrumentation (step 1b of Figure 4). We inform the analyst's tool of the signal or exception and of the newly discovered handler code so that it can add its own instrumentation. We then return control

to the OS, which invokes the program's exception handler. When the handler is done executing, our exit-point instrumentation triggers a callback to our analysis engine (steps 3a-3b of Figure 4), where we check for modifications to the saved PC value. If we detect a change, we analyze the code at the new address, instrument it, and allow the analyst to insert additional instrumentation.

8 Experimental Results

We evaluated our techniques by implementing them in SD-Dyninst and applying it to real and synthetic malware samples. We show that we can efficiently analyze obfuscated, packed, and self-modifying code by applying our techniques to the binary packer tools that are most heavily used by malware authors, comparing these results to two of the most efficient existing tools. We demonstrate the usefulness of our techniques by using SD-Dyninst to create a *malware analysis factory* that we apply to a large batch of recent malware samples. Our analysis factory uses instrumentation to construct annotated program CFG's and a stackwalk at the program's first socket communication.

8.1 Analysis of Packer Tools

Packed binaries begin their execution in highly obfuscated metacode that is often self-modifying and usually unpacks itself in stages. The metacode decompresses or decrypts the original program's payload into memory and transfers control to the payload code at the original program's entry point.

Table 1 shows the results of applying our techniques to the packer tools that are most often used to obfuscate malicious binaries, as determined by Panda Research for the months of March and April 2008, the latest dates for which such packer statistics were available [10]. We do not report results on some of these packers because they incorporate anti-tampering techniques such as self-checksumming, and SD-Dyninst does not yet incorporate techniques for hiding its use of dynamic instrumentation from anti-tampering. We excluded NullSoft's installer tool (with 3.58% market share) from this list because it can be used to create binaries with custom code-unpacking algorithms; though we can handle the analysis-resistance techniques contained in most NullSoft-based packers, we cannot claim success on all of them based on successful analysis of particular packer instances. We also excluded the teLock (0.63% market share) and the Petite (0.25% market share) packer tools, with which we were unable to produce working binaries. The total market share of the packer tools listed by Panda Research is less than 40% of all malware, while at least 75% of malware uses some packing method [8,50]. This discrepancy is a reflection both of the increasing prevalence of custom packing methods and a limitation of the study, which relied on a signature-based tool to recognize packer metacode [11]; most custom packers are derivatives of existing packer tools, which are often modified with the express purpose of breaking signature-based detection.

In Table 1 we divide the execution time of the program into pre- and post-payload execution times, representing the time it takes to execute the binaries'

Table 1. Our analysis techniques applied to the most prevalent packer tools used to obfuscate malware. We analyzed all of the packed binaries that do not employ anti-tampering techniques.

Packer	Malware market share	Over-writes code	Anti tamper-ing	Time (seconds)				
				Pre-exec'n	Pre-payload		Post-payload	
					instr.	uninstr.	instr.	uninstr.
UPX	9.45%			23.44	0.50	0.02	2.80	0.02
PolyEnE	6.21%			22.55	1.24	0.02	2.81	0.02
EXECryptor	4.06%	yes	yes					
Themida	2.95%	yes	yes					
PECompact	2.59%	yes	yes	22.81	3.16	0.02	2.81	0.02
UPack	2.08%	yes		22.56	23.50	0.03	4.08	0.02
nPack	1.74%			23.21	1.54	0.02	2.80	0.02
ASPack	1.29%	yes		22.58	4.42	0.02	2.81	0.02
FSG	1.26%			22.53	1.38	0.03	2.78	0.02
Nspack	0.89%			22.52	2.69	0.03	2.78	0.02
ASProtect	0.43%	yes	yes					
Armadillo	0.37%	yes	yes					
Yoda's Protector	0.33%	yes	yes					
WinUPack	0.17%	yes		22.44	23.60	0.03	4.10	0.02
MEW	0.13%			22.56	3.87	0.03	2.80	0.02

metacode and payload code, respectively. In the uninstrumented case, we determine the proper time split by using a priori knowledge of the packed program to breakpoint its execution at the moment that control transfers from its metacode to its payload code. In the instrumented case, our code-discovery instrumentation automatically identifies this transition by capturing the control transfer to the payload code. We report on SD-Dyninst's pre-execution cost separately, as one of the major benefits of incorporating static analysis techniques into our approach is that we are able to frontload much of the analysis of the program so that it does not affect the program's execution time.

The most striking differences in Table 1 are in the pre-payload cost incurred by SD-Dyninst from one packer to the next. These differences are proportional to the number of occasions in which we discover and analyze new code in the metacode of these binaries. Our instrumentation of the UPX, PolyEnE, nPack, and Nspack packers caused little slowdown in their execution, as their metacode is static and not obfuscated in ways that substantially limit our ability to parse them prior to their execution, while the FSG, MEW, ASPack, UPack, and WinUpack packers are more heavily obfuscated and unpack in stages, requiring that we analyze their code incrementally. The largest contributor to the incremental analysis cost is SD-Dyninst's current inability to resolve the targets of indirect control transfers at parse time, coupled with a simplifying implementation decision to instrument whole functions at a time, meaning that discovery of a new basic block currently causes its entire function to be re-instrumented. SD-Dyninst's performance will improve significantly in the near future through

Table 2. A comparison of pre-payload execution times and instrumented program locations in SD-Dyninst, Renovo, Saffron, and EtherUnpack on packed executables.

	Pre-payload time				Instrumented locations		
Packer	SD-D.	Ren.	Saff.	Ether	SD-D.	Ren.	Saff.
UPX	0.5	5	2.7	7.6	6	2,278	4,526
ASPack	4.4	5	fail	18.7	34	2,045	4,141
FSG	1.6	8	1.4	31.1	14	18,822	31,854
WinUpack	23.6	8	23.5	67.8	23	18,826	32,945
MEW	4.0	6	fail	150.5	22	21,186	35,466

the addition of code-slicing capabilities to Dyninst, which will allow SD-Dyninst to resolve many indirect control transfers at parse time.

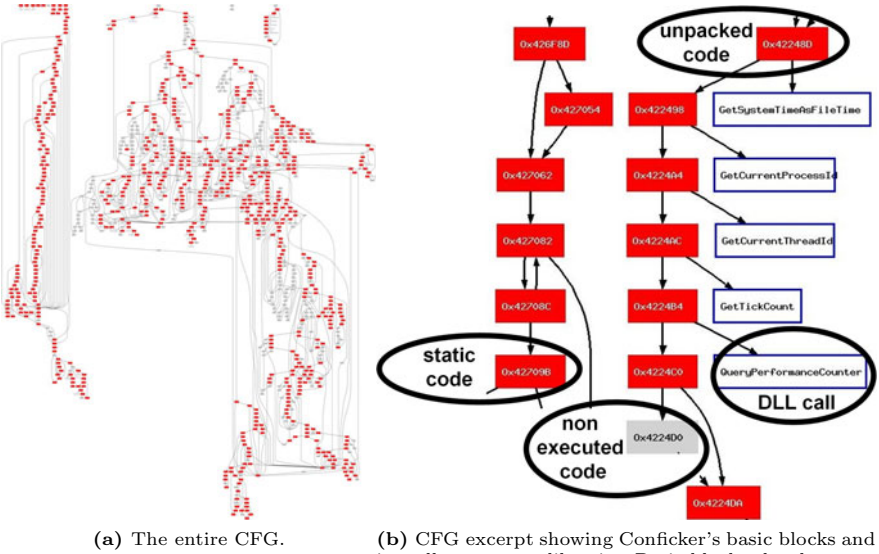
In Table 2 we compare the overall expense of our techniques to the most efficient tools for identifying dynamically unpacked and modified code, Renovo [23], "Saffron for Intel PIN" [41], and EtherUnpack [17]. We executed Saffron and EtherUnpack on our own hardware, but the Renovo tool is not yet publicly available, so we compared against Renovo's self-reported execution times for packed notepad.exe executables, limiting this comparison to the top packer tools that they also analyzed in their study. We ran SD-Dyninst and Saffron on an Intel Core 2 Duo T2350 1.6GHz CPU with 2.5GB of memory, while Renovo's study was conducted on an Intel Core 2 Duo E6660 2.4GHz CPU with 4GB memory and EtherUnpack executed on an Intel Xeon E5520 2.27GHz CPU with 6GB of memory. These numbers reflect the post-startup time it took for SD-Dyninst, Renovo, Saffron, and EtherUnpack to execute the instrumented metacode of the various packer tools.

As seen in Table 2, except in the case of our unoptimized analysis of the WinUpack executable, our pre-payload execution times are comparable to those of Renovo, Saffron, and EtherUnpack *despite the fact* that our tool also analyzes the code while the other tools only identify its dynamically unpacked portions. The Saffron unpacker only partially unpacked the ASPack and MEW executables, as it quits at the first occurrence of written-then-executed code.

The savings afforded by our use of analysis-guided instrumentation help to amortize our analysis costs. Saffron instruments the program at every instruction, while Renovo instruments at all control transfers and at all write instructions (EtherUnpack's single-step mechanism does not rely on instrumentation). We estimated Saffron's use of instrumentation by modifying its source code to maintain a count of unique instrumented instructions, and estimated Renovo's use of instrumentation based on their algorithm description, by instrumenting the packed programs to maintain a count of unique control transfer and write instructions. Table 2 shows that our structural analysis allows us to instrument the program at fewer than a 100th of the locations instrumented by Saffron and Renovo, because our structural analysis allows us to limit our use of instrumentation to instructions whose targets are statically unresolved.

8.2 Malware Analysis

Accomplishing an analysis task using SD-Dyninst requires no more skill from the analyst than performing the same task with Dyninst on a conventional binary. We wrote a malware analysis factory that uses SD-Dyninst to perform code-coverage of malicious program executions by instrumenting every basic block in the program, (both statically present and dynamically unpacked blocks) and removing the instrumentation once it has executed. This instrumentation code consists of only fifty lines. Our factory halts the malware at the point that it attempts its first network communication, exits, or reaches a 30-minute timeout. At this time, the factory prints out a traversal of the program's call stacks and outputs a CFG of the binary that identifies calls to Windows DLL functions and is annotated to



(a) The entire CFG.

(b) CFG excerpt showing Conficker's basic blocks and its calls to system libraries. Basic blocks that have executed are shown in red blocks, non-executed blocks are in light-grey, and API functions are shown as white rectangles.

Fig. 5. Two views of Conficker A's control flow graph. The CFG in part (a) can be explored in an interactive viewer, as shown in the excerpt from part (b).

```
top  pc=0x7c901231 DbgBreakPoint_0x7c901230 in ntdll.dll    [Win DLL]
     pc=0x10003c83 DYNbreakPoint_0x10003c70 in dyn_RT.dll   [Instrument.]
     pc=0x100016f7 DYNstopThread_0x10001670 in dyn_RT.dll   [Instrument.]
     pc=0x71ab2dc0 select_0x71ab2dc0        in WS2_32.dll   [Win DLL]
base pc=0x401f34   func=nosym1f058_0x41f058 in cf.exe       [Conficker]
```

Fig. 6. An SD-Dyninst stack walk taken when the Conficker A binary executes Winsock's `select` routine. The stack walk includes frames from our instrumentation, `select`, and Conficker.

distinguish between blocks that have executed and those that have not. If the malware fails to send a network packet, we verify our analysis by comparing against a trace of the malware's execution to ensure that our analysis reaches the same point.

We set up our malware analysis factory on an air-gapped system with a 32-bit Intel-x86 processor running Windows XP with Service Pack 2 inside of VMWare Server. We then analyzed 200 malware samples that were collected by Offensive Computing [3] in December 2009. Our tool detected code unpacking in 27% of the samples, code overwrites in 16%, and signal-based control flow in 10%. 33% of the malicious code analyzed by our hybrid techniques was not part of the dynamic execution trace and would not have been identified by dynamic analysis. For the malware samples that attempted to communicate with the network, our analysis factory walked their call-stacks to identify the code in the malicious executable that triggered the network communication.

As an example of the kinds of results produced by our factory, in Figures 5 and 6 we show two of its analysis products for the Conficker A malware binary. Our factory created similar analysis products for all the other malware binaries that we analyzed, and these can be viewed online at `http://www.paradyn.org/SD_Dyninst/`. In Figure 5a we show the annotated CFG of the Conficker A binary in its entirety, while Figure 5b shows an excerpt of that graph, highlighting the fact that SD-Dyninst has captured static and dynamic code, both code in the executable and code in Windows DLL's, and both code that has executed and code that has not executed but that may be of interest to the analyst. Figure 6 shows our traversal of Conficker's call stacks at its first call to the `select` routine. As seen in this stack trace, we are able to identify the stack frames of functions that lack symbol information, an important benefit of our analysis capabilities. While existing stackwalking techniques are accurate only for statically analyzable code [26], our hybrid analysis enables accurate stackwalking by virtue of having analyzed all of the code that could be executing at any given time.

9 Conclusion

We create a hybrid analysis algorithm that makes it possible to analyze and control the execution of malicious program binaries in a way that is both more intuitive and more efficient than existing methods. Our combination of static and dynamic analysis allows us to provide analysis-guided instrumentation on obfuscated, packed, and self-modifying program binaries for the first time. We implemented these ideas in SD-Dyninst, and demonstrated that they can be applied to most of the packing tools that are popular with current malware. We demonstrated the usefulness of our techniques by applying SD-Dyninst to produce analysis artifacts for the Conficker A binary that would have required substantial manual effort to produce through alternative methods.

Ongoing research in the Dyninst project promises to address the two primary limitations of this work. First, our instrumentation's changes to the program's address space can be detected through anti-tampering techniques such as self-checksumming. The second problem is that our parsing techniques assume that

the targets of conditional control transfers always represent real code, meaning that the obfuscation proposed by Collberg et al. [14] could be used to pollute our parse with non-code bytes, potentially causing us to instrument data and causing the program to malfunction. Both of these problems are being addressed by a piece of ongoing research in Dyninst that will ensure that the presence of instrumentation will not impact a program's data accesses.

Two additional analysis-resistance techniques that merit discussion are anti-debugging and timing checks. Several Windows API and system calls can be used to detect the presence of a debugger. Invocations of such functions are easily detected by SD-Dyninst instrumentation and we have disabled those we have come across, but from the literature [18] we know there are additional anti-debugging methods that our current implementation does not disable. Timing checks similarly depend on Windows API calls, system calls, and special-purpose instructions that are easily detected by SD-Dyninst. However, we have not had to disable timing checks, as most existing checks are tuned to detect the significant slowdowns incurred by single-step debugging techniques and are not triggered by our more efficient instrumentation-based techniques. It is possible that future timing checks could detect the slowdown caused by our algorithm's online analysis, in which case we could adapt Ether's [17] clock emulation techniques to hide the slowdown from the program.

Acknowledgments

This work is supported in part by Department of Energy grants DE-SC0004061, 08ER25842, 07ER25800, DE-SC0003922, Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL), and National Science Foundation Cybertrust grants CNS-0627501, and CNS-0716460.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Computer economics 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code (2007)
2. Darkparanoid virus (1998)
3. Offensive computing, `http://www.offensivecomputing.net`
4. Anckaert, B., Madou, M., Bosschere, K.D.: A model for self-modifying code. In: Information Hiding, Alexandria, VA, pp. 232–248 (2007)
5. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: International Conference on Compiler Construction, New York, NY, pp. 5–23 (2004)
6. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology 2(1), 66–77 (2006)
7. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, Anaheim, CA, pp. 41–46 (2005)
8. BitDefender: BitDefender anti-virus technology. White Paper (2007)
9. Bustamante, P.: Malware prevalence. Panda Research web article (2008)
10. Bustamante, P.: Packer (r)evolution. Panda Research web article (2008)

11. Bustamante, P.: Personal correspondence (2009)
12. Chiang, K., Lloyd, L.: A case study of the rustock rootkit and spam bot. In: First Conference on Hot Topics in Understanding Botnets, Cambridge, MA (2007)
13. Cifuentes, C., Emmerik, M.V.: UQBT: adaptable binary translation at low cost. Computer 33(3), 60–66 (2000)
14. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Symposium on Principles of Programming Languages, San Diego, CA, pp. 184–196 (1998)
15. Coogan, K., Debray, S., Kaochar, T., Townsend, G.: Automatic static unpacking of malware binaries. In: Working Conference on Reverse Engineering, Antwerp, Belgium (2009)
16. Danehkar, A.: Inject your code into a portable executable file (2005), `http://www.codeproject.com/KB/system/inject2exe.aspx`
17. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: Conference on Computer and Communications Security, Alexandria, VA (2008)
18. Ferrie, P.: Anti-unpacker tricks. In: International CARO Workshop. Amsterdam, Netherlands (2008)
19. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Network and Distributed System Security Symposium, San Diego, CA (2003)
20. Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 98–115. Springer, Heidelberg (2008)
21. Hind, M., Pioli, A.: Which pointer analysis should I use? In: International Symposium on Software Testing and Analysis, Portland, OR, pp. 113–123 (2000)
22. Hollingsworth, J.K., Miller, B.P., Cargille, J.: Dynamic program instrumentation for scalable performance tools. In: Scalable High Performance Computing Conference, Knoxville, TN (1994)
23. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Workshop on Recurring Malcode, Alexandria, VA (2007)
24. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: USENIX Security Symposium, San Diego, CA (2004)
25. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Conference on Computer and Communications Security, Washington, DC, pp. 290–299 (2003)
26. Linn, C., Debray, S., Andrews, G., Schwarz, B.: Stack analysis of x86 executables (2004) (manuscript)
27. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Programming Language Design and Implementation, Chicago, IL, pp. 190–200 (2005)
28. Madou, M., Anckaert, B., de Sutter, B., Bosschere, K.D.: Hybrid static-dynamic attacks against software protection mechanisms. In: ACM Workshop on Digital Rights Management, Alexandria, VA, pp. 75–82 (2005)
29. Maebe, J., Bosschere, K.D.: Instrumenting self-modifying code. In: International Workshop on Automated and Algorithmic Debugging, Ghent, Belgium (2003)
30. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Annual Computer Security Applications Conference, Miami Beach, FL (2007)

31. Mirgorodskiy, A.V., Miller, B.P.: Autonomous analysis of interactive systems with self-propelled instrumentation. In: International Conference on Parallel Computing, San Jose, CA (2005)
32. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Symposium on Security and Privacy, Oakland, CA, pp. 231–245 (2007)
33. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Annual Computer Security Applications Conference, Miami Beach, FL (2007)
34. Nanda, S., Li, W., Lam, L.C., Cker Chiueh, T.: Bird: Binary interpretation using runtime disassembly. In: International Symposium on Code Generation and Optimization (CGO 2006), New York, NY, pp. 358–370 (2006)
35. Neumann, R.: Exepacker blacklisting part 2. Virus Bulletin pp. 10–13 (2007)
36. Nguyen, A.M., Schear, N., Jung, H., Godiyal, A., King, S.T., Nguyen, H.: Mavmm: A lightweight and purpose-built vmm for malware analysis. In: Annual Computer Security Applications Conference, Honolulu, HI (2009)
37. Perriot, F., Ferrie, P.: Principles and practise of x-raying. In: Virus Bulletin Conference, Chicago, IL, pp. 51–66 (2004)
38. Popov, I., Debray, S., Andrews, G.: Binary obfuscation using signals. In: USENIX Security Symposium, Boston, MA, pp. 275–290 (2007)
39. Porras, P., Saidi, H., Yegneswaran, V.: A multi-perspective analysis of the storm (peacomm) worm. SRI International Technical Report (2007)
40. Porras, P., Saidi, H., Yegneswaran, V.: An analysis of conficker's logic and rendezvous points. SRI International Technical Report (2009)
41. Quist, D., Ames, C.: Temporal reverse engineering. In: Blackhat, USA, Las Vegas, NV (2008)
42. Quist, D.A., Liebrock, L.M.: Visualizing compiled executables for malware analysis. In: Workshop on Visualization for Cyber Security, Atlantic City, NJ (2009)
43. Rosenblum, N.E., Zhu, X., Miller, B.P., Hunt, K.: Learning to analyze binary computer code. In: Conference on Artificial Intelligence, Chicago, IL (2008)
44. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In: Annual Computer Security Applications Conference, Miami Beach, FL, pp. 289–300 (2006)
45. Security, P.: Annual report Pandalabs (2008)
46. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Network and Distributed System Security Symposium, San Diego, CA (2008)
47. Sites, R.L., Chernoff, A., Kirk, M.B., Marks, M.P., Robinson, S.G.: Binary translation. Communications of the ACM 36(2), 69–81 (1993)
48. Srivastava, A., Edwards, A., Vo, H.: Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50 (2001)
49. Srivastava, A., Eustace, A.: ATOM: a system for building customized program analysis tools. In: Programming Language Design and Implementation, Orlando, FL (1994)
50. Trilling, S.: Project green bay–calling a blitz on packers. In: CIO Digest: Strategies and Analysis from Symantec, p. 4 (2008)
51. Vigna, G.: Static disassembly and code analysis. In: Malware Detection. Advances in Information Security, vol. 35, pp. 19–42. Springer, Heidelberg (2007)
52. Yegneswaran, V., Saidi, H., Porras, P.: Eureka: A framework for enabling static analysis on malware. Technical Report SRI-CSL-08-01 (2008)

Anomaly Detection and Mitigation for Disaster Area Networks

Jordi Cucurull, Mikael Asplund, and Simin Nadjm-Tehrani

Department of Computer and Information Science, Linköping University
SE-581 83 Linköping, Sweden
{jordi.cucurull,mikael.asplund,simin.nadjm-tehrani}@liu.se

Abstract. One of the most challenging applications of wireless networking are in disaster area networks where lack of infrastructure, limited energy resources, need for common operational picture and thereby reliable dissemination are prevalent. In this paper we address anomaly detection in intermittently connected mobile ad hoc networks in which there is little or no knowledge about the actors on the scene, and opportunistic contacts together with a store-and-forward mechanism are used to overcome temporary partitions. The approach uses a statistical method for detecting anomalies when running a manycast protocol for dissemination of important messages to k receivers. Simulation of the random walk gossip (RWG) protocol combined with detection and mitigation mechanisms is used to illustrate that resilience can be built into a network in a fully distributed and attack-agnostic manner, at a modest cost in terms of drop in delivery ratio and additional transmissions. The approach is evaluated with attacks by adversaries that behave in a similar manner to fair nodes when invoking protocol actions.

1 Introduction

Disaster area networks are created through spontaneous mobilisation of ad hoc communication when the existing infrastructure is wiped out or severely overloaded. In such environments, in addition to local establishments of cellular networks and satellite connections there is a potential for hastily formed networks (HFN) [1] of wireless devices connecting via 802.11 or similar technologies. One of the major needs in a disaster area is timely dissemination of information destined for a large group of actors. However, due to the nature of the multi-party engagements, and massive engagement of volunteers there is little room for establishment of mutual trust infrastructures. Any dissemination protocols destined for such environments require to function in a chaotic context where the node identifiers or even the number of involved nodes cannot be assumed.

The physical aspects of above networks can be characterised by intermittent connectivity, leading to networks in which existence of partitions is a norm. This creates a variation of mobile ad hoc networks (MANET) with no contemporaneous routes among the nodes, also referred to as intermittently connected MANET (IC-MANET). Experience from the Katrina HFN [2] shows that even

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 339–359, 2010.

in disaster environments there are security threats – actors who try to disrupt operations or use the information for own benefit. However, the application of security measures in these environments is far from trivial. This paper addresses security issues that impact dissemination of messages, and thereby focuses on the availability of the dissemination services in IC-MANET.

We study the impact of intrusions on a dissemination protocol, Random Walk Gossip (RWG) that is designed to run in IC-MANET in a disaster area network [3]. This algorithm is intended to disseminate important messages to any k receivers, thereby a manycast algorithm that does not rely on knowledge about the network nodes. Its main characteristics are that it overcomes lack of information with opportunistic contacts and uses a store-and-forward mechanism to overcome network partitions. Moreover, to act in an energy-efficient manner it will try to avoid excessive transmissions by decentralised control of number of custodians for a given message.

In such a scenario the adversary has no choice other than behaving in a way that resembles the rest of the nodes in the network, and we further assume that the adversary too needs to be efficient in its use of energy and bandwidth resources. In fact the adversary may not act normally, but seen from a restricted view, being the local knowledge at each mobile node, it follows fragments of the protocol specification.

Intrusion detection mechanisms are intended to identify malicious activity targeted at the resources of a monitored system, broadly classified as misuse or anomaly detection. The former requires the formulation of misuse constraints, which are extremely complex when the adversary behaves within the boundaries of the protocol specifications and when at the same time it must be suitable for different environments, i.e. dynamic load, partition sizes, varying local densities and connectivity changes. In the IC-MANET context anomaly detection is a suitable approach to intrusion detection while misuse detection is less appropriate. First, the fact that the adversary behaves in a similar way to the fair nodes makes formulation of misuse constraints hard if not impossible. Second, even if we can formulate a set of rules for undesirable packet sequences, these will hardly work in all nodes due to dynamic load and partition changes.

Our approach builds on a learning based anomaly detection technique that uses only normal data in the learning phase. While this might be a limitation of the approach, since there is no guarantee that attacks are not present in the early deployment phase in the scenario, we believe that the efficiency of the technique will outweigh its disadvantages. Another major problem in highly unpredictable and partitionable networks is what to do when an attack is suspected. If the network is generally chaotic and the goal is to maintain dissemination levels then it is less relevant to exactly identify adversary nodes and try to isolate or ignore them. We therefore suggest mitigation approaches that enable each node to adjust its own behaviour thereby reducing the effect of the suspected attack.

The threat model that we consider is that the adversary tries (1) to drain the network resources, both at node level (battery life) and network level (available bandwidth), thereby reducing the dissemination flows, and (2) acts as an

absorbing patch which reduces some message dissemination in its vicinity, acting as a grey hole at certain times and locations. Clearly this threat model creates a challenging type of adversary to detect.

Our detection and mitigation approach has been evaluated in a simulation setting where an implementation of the RWG algorithm is running with a disaster mobility model adapted from earlier work [4]. The evaluations indicate that the approach indeed creates a resistance to attacks that match the above threat model, and show that the network recovers from the attack when it is of a transient nature. Moreover, our approach dampens the effect of the attacks on the network resources by preserving the overall overhead in the network compared to the non-mitigated case, whilst not losing the delivery goals significantly. These results are obtained despite the fact that the classical metrics used for evaluation of intrusion detection do not show good results. The paper discusses why the network performance metrics are more useful in IC-MANET clarifying the impact of partitions, traffic load and the store-and-forward approach.

The contributions of the paper are as follows:

- Presentation of a scalable approach to anomaly detection and mitigation in partitionable ad-hoc networks with scarce resources that run a given dissemination protocol suitable for these environments. The detection algorithm is scalable since it is fully distributed and efficient. It is a statistical mechanism reminiscent of the chi-square technique [5]. It has been adapted to the specific RWG protocol by selection of appropriate (general) features.
- Illustration of the approach using a simulation platform, and specifically analysing why the performance based metrics outperform the classic detection rate and false positive rate metrics in such disaster area networks.

2 Related Work

Yang *et al.* [6] describe the major security challenges in MANET and some of the existing solutions. Among the identified challenges are the lack of a well-defined place to deploy the security solutions, the vulnerability of the information contained within the devices, the fact of communicating within a shared medium, bandwidth, computation and energy resource constraints, the dynamic network topology, and the wireless channel characteristics (e.g. interference and other effects leading to unreliability). It is also stated that a complete security solution should integrate prevention, detection and reaction components. Prevention typically evolves around establishment of trust, often based on cryptographic techniques. However, trust is not easy to achieve in such scenarios [1] and cryptographic techniques, as studied in Prasithsangaree and Krishnamurthy [7], usually are computationally too expensive. Farrell and Cahill [8], in the context of delay-tolerant networks, also mention the lack of cryptographic key management schemes as an open issue.

Orthogonal to the preventive perspective we need to consider the role of intrusion detection in IC-MANET. Several approaches to intrusion detection have

already been proposed for the MANET Ad hoc On Demand Distance Vector (AODV) and Optimised Link State Routing (OLSR) protocols. However, to our knowledge no earlier works address multicast protocols, and specifically not those suitable to run in a partitioned MANET. Some authors propose that specifying, distributing and updating the signatures of the attacks is not feasible [9] in these environments. Instead, anomaly detection is easier to apply since the normality model can be created and updated in each node from its own observations. Hence, abnormal behaviours in the specific context of a particular node, even if they are within the boundaries of the protocol specifications, can be detected. Garcia-Teodoro *et al.* [10] present an extensive review of the most popular techniques used in anomaly detection, which can be roughly classified into statistical based, knowledge based, and machine learning based. The most significant challenges of anomaly detection are also mentioned, namely low detection efficiency and high false positive rate, low throughput and high cost, absence of appropriate metrics and assessment methodologies for evaluating and comparing different techniques, protection of the intrusion detection mechanism from attacks, and analysis of ciphered data. In our work we confirm that the classic metrics used in wired or fully connected wireless networks are not appropriate in IC-MANET. We believe comparisons on common simulation platforms (as long as the authors make their code accessible to other researchers) is a first step for comparative evaluation.

Although anomaly detection for IC-MANET has to be geared towards protocols that in fact manage the challenges of multiple partitions – what we aim to address in this paper – we would like to name a few precursors for anomaly detection in MANET. Nakayama *et al.* [11] propose an adaptive method to detect attacks on the AODV routing protocol. The method is based on the calculation of projection distances using multidimensional statistics and the Principal Component Analysis (PCA) to determine the axis that expresses the most relevant data correlations. Cabrera *et al.* [12] propose a method to detect attacks on AODV and OLSR protocols. The method consists of three hierarchical and distributed intrusion detection modules based on pattern recognition and classifier fusion. Liu *et al.* [9] too present a method to detect attacks on the AODV routing protocol. The method is based on the combination of two data mining approaches over data obtained from the MAC and network layers. The technique allows the identification of the MAC address of the attacker, although it can be easily spoofed. Later, a Bayesian method is used to correlate local and global alerts. It also proposes a collaborative decision mechanism among nodes in the radio transmission range. These approaches are not applicable to our problem area. First due to the manycast nature of dissemination and secondly due to the intermittent connectivity in disaster area networks.

Among the few works that address intrusion detection in IC-MANET, there is work by Chuah *et al.* [13] proposing a rule-based misuse detection mechanism targeted towards delay-tolerant networks. It builds on a history-based routing protocol, detects attacker nodes, and mitigates their effects by keeping them on a black list, i.e. through isolation. Other security-related work in IC-MANET is

concerned with prevention, e.g. Scalavino *et al.* [14] who propose an authorisation scheme to protect the data exchanged in disaster events.

A main challenge of anomaly detection in MANET [9] is that most of the approaches do not succeed with localisation of the attack sources. There is no prior trust relationship among the network nodes, and network operation relies on altruism and cooperation of neighbour nodes. Also the fact that nodes fail to respond, e.g. through network congestion, link failure, or topology changes, can be confused with intrusions [15], producing high false positive rates.

The metrics used on intrusion detection in MANET are usually based on the accounting for detection rate and false positive rate typically presented as ROC curves. In most cases these metrics reflect the number of attacks detected, but sometimes they show the number of attackers detected [13]. The detection delay is not usually taken into account, but there are a few exceptions [16,17,13]. There are approaches that quantify the impact of the detectors, such as network overhead or the CPU speed-up [18], or data delivery ratio to evaluate the impact of attack response [13]. In this paper we also advocate the use of delivery ratio and total transmissions (as an indicator of overhead) for evaluation purposes.

Finally, response and mitigation of attacks is one of the topics that has not been considered much in the intrusion detection literature for wireless ad-hoc environments. Some MANET works [16,19] just mention it and propose certain actions, but do not really apply it. There are a few exceptions in which mitigation is really applied [13,20]. In addition to the network performance metrics to show the benefits of the approaches, the work by Wang *et al.* [20] also proposes metrics to decide whether it is worth to enable the mitigation or not. This is an interesting direction that should be explored within IC-MANET too, but we postpone it to future works.

3 Protocol Description and Threat Model

3.1 Protocol Description

The Random Walk Gossip (RWG) is a message dissemination protocol for intermittently connected networks that has been presented and evaluated in a previous publication [3]. Here we will try to provide just as much information as needed to understand the threat model that we have used.

The protocol is designed to cope with the challenges faced in disaster area networks such as intermittent connectivity, scarcity of bandwidth and energy, as well as unknown and unpredictable network topologies with partitions. RWG is a manycast protocol, which means that a message is intended to reach a given number k of nodes. When k nodes have been reached, the message is k-delivered and does not need to be propagated anymore, thus not wasting energy. RWG is based on a store-and-forward mechanism, i.e. each node keeps the messages to forward in a local buffer until they have been delivered. This mechanism prevents the loss of messages because of network partitions.

When a message is sent in a connected part of the network, it will perform a random walk over the nodes, until all the nodes in the partition are informed of

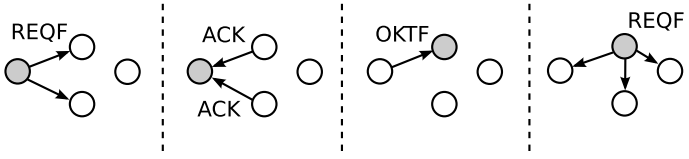


Fig. 1. Random Walk Gossip

this message. This walk is controlled by a three-way packet exchange shown in Figure 1. First a Request to Forward (REQF), that includes the message payload, is sent by the current custodian of the message (indicated as grey in the picture). The neighbouring nodes that hear the REQF reply with an acknowledgement packet (ACK). The custodian chooses one of these nodes randomly and sends an OK to Forward (OKTF) to this node indicating that it will be the next custodian. The other nodes will retain the message without actively disseminating it. They will keep the message as *inactive* until it expires. Partitions can be overcome by the movement of nodes. Thus, new uninformed nodes will be informed by some node that keeps the message as *inactive* and restarts to disseminate. This process will continue as long as no more uninformed nodes remain in the network or the message is k-delivered.

All the packet types share the same header structure. In order to keep track of which nodes have seen a given message, each packet header contains a bit vector called the *informed* vector. When a node receives the message it produces a hash of its own address and puts a 1 in the bit vector in the field corresponding to the hash. This allows the protocol to know when a message is k-delivered, and to tell the potential future recipients of the message how far the message has reached towards its dissemination goal (summing the number of 1's indicates the current known local knowledge on this). The *informed* vector also indicates which nodes have received the message. If a node A hears a new neighbour B, then A will go through the messages stored in its buffer to see if B has not yet been informed of any of the messages, in which case those messages will be reactivated and sent to node B (and other uninformed nodes in the vicinity).

Finally, when a node realises that a message is k-delivered it sends a Be Silent (BS) packet to its vicinity. This packet will cause all receiving nodes to also realise that the message is k-delivered and thus remove it from their buffers. No new BS packets are sent upon the reception of a BS packet.

3.2 Threat Model

Routing and dissemination protocols for MANET are usually based on cooperation of the network nodes and assume fair play. RWG is not an exception and an attacker can take advantage of it. There are many ways a malicious node can attempt to disrupt the dissemination activity in a network. This paper focuses on the mitigation of low-cost attacks which are consistent with the protocol specification. We study how a disrupting node will try to impact the dissemination and resource usage of the network without investing too much of its own energy

resources. Recall that the only packet type with a payload is the REQF packet. This is also the one that claims more in terms of transmission energy and bandwidth. Using forged inexpensive ACKs three aspects of the protocol operation can be attacked:

- **Discovery of new nodes:** RWG discovers new nodes in the vicinity by listening to the packets the node receives. Each time a packet is received the messages stored in the local buffer are checked to see if they have been already delivered to the sender of that packet. If that is not the case they are forwarded to the node. An attacker can exploit this mechanism by sending many ACK packets with different fake sender addresses and create a high number of message retransmissions. The fake addresses are randomly generated and there is no mechanism to prevent their usage.
- **Delivery status propagation:** The propagation of the delivery status of the messages is done through the *informed* vector included in the sent packet headers. An attacker can manipulate these vectors and take advantage of the other nodes to propagate them using ACK packets.
- **Selection of custodians for a given message:** When a message is forwarded to a group of nodes, they answer with an ACK packet. RWG uses these ACK packets to randomly choose one of the nodes as the next custodian of the message. An attacker could exploit this mechanism to be elected as the next custodian by answering with several ACKs increasing the probability of being chosen.

It is assumed that the adversaries will have a complete knowledge of the protocol and that will act according to the RWG specifications. Though our anomaly detection algorithm is oblivious to the attack patterns, we will later use two specific instances of attacks (see Section 5.2) based on exploiting some of the operations described for the purpose of evaluation.

4 Anomaly Detection and Mitigation

Anomaly detection is based on the construction of a model that represents the normal behaviour of a system and which is used to determine abnormal situations. Since MANET are usually operated by resource constrained devices a statistical-based approach has been selected as an anomaly detector since it has a smaller footprint than other techniques.

4.1 Detection Algorithm

The anomaly detector we propose represents normality as a vector of numerical values called features. The algorithm is based on a distance function $D(x_i)$ that calculates sums of squared differences between a given observation x_i of the system (which contains F features) and the normality model $\bar{x}$ to decide if the observation is anomalous or not (see Eq. 1). An observation is obtained and evaluated each time a packet is received. According to the central limit theorem,

if the number of variables is large enough, then the calculated sums of squared differences will follow a normal distribution. Hence a threshold (T_1), based on the statistical three-sigma rule (also called 68-95-99.7 rule) [21], is introduced to determine if the distance measured is outside of the values considered normal. The work flow of the system has two differentiated parts.

$$D(x_i) = \sum_{j=1}^{F} (x_{i,j} - \bar{x_j})^2 \tag{1}$$

1. **Training:** In this part in which the normality model of the system is created, only observations of the normal behaviour of the system are used. The model consists of a vector ($\bar{x}$) with the average value of each feature, two vectors (max, min) with the maximum and minimum values expected for each feature under normal conditions, and a threshold (T_1) that states which is the maximum distance observed from the average field of normality. The normality model is created during a period of time that includes two consecutive steps that comprise N and M observations respectively.

   (a) **Calculation of average, maximum and minimum values:** During a period of time with a number of N observations, the average ($\bar{x}$), maximum (max), and minimum (min) vectors are calculated. The last two vectors are used for normalisation, i.e. to keep all the features from normal observations within the same range of values. Normalisation is also applied to $\bar{x}$.

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{2}$$

   (b) **Calculation of the threshold:** During a period of time, and for a number of M observations, the distance $D(x_i)$ between an observation x_i and the calculated average $\bar{x}$ is measured. T_1 (see Eq. 3, 4, and 5) is defined as the mean of the distances calculated (μ) plus three times their standard deviation (σ). According to the three-sigma rule the range $[\mu - 3\sigma, \mu + 3\sigma]$ should cover 99.7% of the distances of the normal observations evaluated. Note that just the upper limit is used, because evaluations with small distances are not considered anomalous.

$$\mu = \frac{1}{M} \sum_{i=1}^{M} D(x_i) \tag{3}$$

$$\sigma = \sqrt{\frac{1}{M} \sum_{i=1}^{M} (\mu - D(x_i))^2} \tag{4}$$

$$T_1 = \mu + 3\sigma \tag{5}$$

2. **Testing:** During this step the detector is fed with observations of the system behaviour that can be anomalous. The detector decides whether an observation x_i is anomalous by calculating the distance $D(x_i)$ from $\bar{x}$, which determines how far is the current observation from the normal behaviour of the system, and compares it with T_1. If $D(x_i) > T_1$ the observation is categorised as anomalous, and if $D(x_i) \leq T_1$ it is categorised as normal.

4.2 Features

The features of an anomaly detector are the variables which are believed to characterise the behaviour of the monitored system. Our approach uses features at the routing layer and most of them are statistical.

- **Packet rates:** Number of packets of each type received during the last I_1 seconds. There are four of these features, one for each packet type.
- **Packet distances:** Distance, measured in number of packets received, between the reception of two specific types of packets. E.g., number of packets received between the reception of a REQF and the next ACK. There are sixteen of these features that cover all the possible packet type combinations.
- **Packet rate differences:** Relative difference in the packet rates calculated for each type of packet. There are six features, one for each relevant combination.
- **Number of different source addresses:** Number of different source addresses counted in the packets received during the last I_2 seconds.
- **Packet ratios:** Quotient of the number of packets received of a specific type compared to another packet type among the last I_3 packets received. There are three of these features: ACK/REQF, ACK/OKTF, ACK/BS.
- **Summation of informed vectors:** Summation of all the positions of the *informed* vectors received in the last I_4 packets.

Because the evaluation is carried out each time a packet is received, the features that provide information about the last packets received are implemented as sliding windows over the intervals I_1, I_2, I_3, and I_4.

4.3 Alert Aggregation

Statistical anomaly detection requires a certain time to detect an anomaly within the system. As alerts cannot be mapped to the specific packets causing the attacks, the alarms must be raised after an interval of suspicion. This is the reason why the alerts raised by the detector are processed and aggregated during an interval I_a of aggregation.

In each of these periods the number of packets evaluated and the number of alerts registered are counted. Then, an alarm is raised if the number of alerts within that period exceeds a certain threshold (T_2). The threshold is a tunable parameter of the system which is defined in terms of proportion of alerts registered over the number of packets evaluated during I_a.

4.4 Mitigation

When an alarm is raised in a node the mitigation scheme is locally enabled. As it will be explained in Section 5.3, a careful RWG operational mode is proposed to cover the possible attacks that fall within the threat model defined. Since it is not clear whether an attack is transient, continuous or intermittent, we need to decide how long a mitigation should take place. In this paper we have simply evaluated a mitigation that takes place over a constant interval I_m ($> I_a$). This prevents the system from disabling the mitigation too early as a consequence of the beneficial effects of the mitigation instead of the finalisation of the attack.

5 Evaluation

This section evaluates the detection and mitigation approach applied to RWG in a disaster area scenario against the threat model described in Section 3.2.

5.1 Simulation Setup

The performance of the approach has been evaluated using the Network Simulator 3 (ns-3) with an implementation of the detection and mitigation mechanisms embedded in the RWG protocol implementation at the network layer.

The disaster area scenario includes the mobility traces from Aschenbruck *et al.* [4], based on a large training manoeuvre in preparation of the FIFA world cup in Germany in 2006. The original traces include 150 mobile nodes. To induce partitions and create an intermittently connected network we have selected 25 of the nodes, chosen uniformly over the locations in the area, while maintaining the trace for that node. This creates a similar network with lower density. Five other nodes have been chosen as attackers while acting in the same network, again with a uniform distribution among the fair nodes. The attacker nodes do not create normal traffic (data) in the network, but produce packets that are compatible with the protocol specification as described in section 5.2. This is aligned to the threat model defined, where the attacker spend the minimum energy possible. All the nodes use the 802.11a protocol, at 6Mb/s data rate with a radio range of 24 meters. The speed of the nodes varied in the range 1-2 m/s in an area of 200m x 350m. The load is generated by introducing in the network a total of 15 messages to disseminate every second from randomly chosen nodes. Each message has set to be delivered to minimum number of 10 nodes ($k = 10$).

The simulation time for each run is 3000 seconds. The first 200 seconds are discarded (not used for anomaly detection) due to start-up time for the protocol. The following 1400 seconds are used for training the system (half of them for calculating $\bar{x}$, min and max vectors and the rest for the threshold), and the last 1400 seconds are used for evaluation. Each simulation is repeated 10 times with different sets of traces and all the results shown are averages over these 10 runs. The alert aggregation window (I_a) is chosen as 10 seconds (unless otherwise stated). The selected threshold (T_2) for the alert aggregation process is set up to

30%. The mitigation period (I_m), during which a mitigation remains enabled is set up to 200 seconds. The intervals used to calculate the features (I_1, I_2, I_3, and I_4) are set up to 5 seconds, 10 seconds, 50 packets, and 100 packets, respectively.

5.2 Generated Attacks

To show the effectiveness of the detection and mitigation approach, two attacks that fall into the threat model described in Section 3.2 have been implemented.

- **Draining attack:** It makes the nodes around the attacker to transmit more packets than usual in order to drain their batteries and waste some bandwidth. The effect, that exploits the RWG node discovery mechanism, is achieved by regularly sending ACK packets with different fake identities. As it is depicted on Fig. 2 the affected neighbours (A and B affected by C in the example) respond to each ACK by sending all the messages stored in their buffers ($m1$, $m2$, $m3$) which are in *inactive* state, since the identity announced in the ACK (n_F) is completely new and it seems to come from a not yet informed node. The attack is cheap since just one inexpensive ACK packet issued by the attacker may reach several nodes which can answer with several possibly expensive REQF packets that, besides, induce other responses to them (3 REQF, 3 ACK and 3 OKTF in the example).

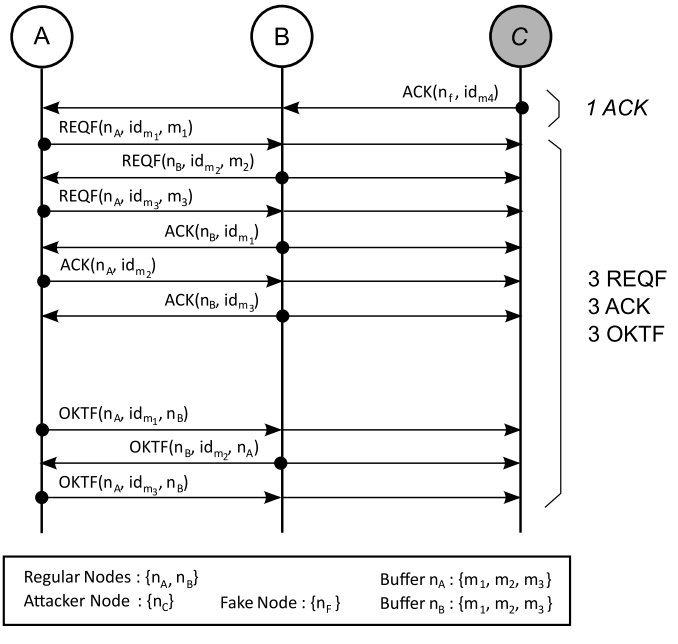


Fig. 2. Draining attack

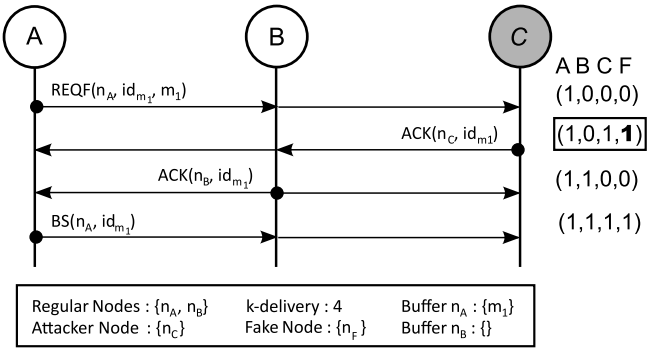


Fig. 3. Grey hole attack

– **Grey hole attack:** This attack, which exploits the propagation of the message delivery status, consists of making the nodes around the attacker to believe that the messages they disseminate have already reached k nodes as required. This makes the fair nodes to execute the mechanisms for removing the message, thus resulting in a reduction of network message k-delivery ratio. As can be seen in Fig. 3 the attacker answers the REQF packets received with an ACK that contains a forged *informed* vector (see values within parenthesis in the example). The vector is modified to include $k-1$ bits set to 1. Hence, when another fair ACK is received the node which has sent the REQF considers that the message has been disseminated to k nodes and issues a BS packet. Note that the attacker does not directly set the number of bits of the *informed* vector to k in order to go unnoticed.

In both cases the adversaries do not participate in the normal operation of the network, but can listen and send packets as any other node. Both of the attacks are tested in both continuous and transient modes. The continuous mode enables the attack during 2/3 of the detection time. From the 2067th time step in our tests, until the end of the simulation. While the transient mode enables the attack during a certain interval of the simulation, from 2200 to 2400 seconds in our tests. The former shows the effects of a persistent attack, while the latter shows the effects of an attack that disappears shortly after.

These attacks have indeed a significant impact on network performance. Beginning with the draining attack, it is performed by 5 nodes each sending 10 ACK packets/second with different identities. Each of these ACKs produces around 15 direct responses as REQF packets issued by the victims. The impact of the continuous draining attack can be seen on Fig. 4, where a huge and sharp increase of the network packet transmissions can be observed soon after the attack. Note that a peak, with around 150% higher packet transmission rate, is registered during the first 100 seconds of the attack. Later this rate is reduced and stabilised to around a 90% higher rate compared to the no attack case. This is due to the fact that just at the beginning of the attack there are more inactive messages ready to be forwarded in the buffers of the fair nodes.

The grey hole attack, whose goal is to reduce the chances of successful dissemination of messages, is performed by 5 nodes each one answering to all the REQF packets they receive with forged ACK packets. The impact of the continuous grey hole attack can be seen in Fig. 6, which depicts how the message k-delivery rate, in comparison with the messages introduced into the network, suddenly drops to a 10% of the normal rate (which in this scenario is around 10 messages/second) just after the beginning of the attack.

5.3 Implemented Mitigations

In a highly unpredictable environment with pockets of connectivity, we need to act in a way that works with unknown node IDs and "fuzzy" normality. Instead of suspecting individual nodes and isolating them (which is very difficult to do accurately) as for example in the work by Wang *et al.* [20], our approach is based on the adjustment of the protocol behaviour in the own node to a careful mode. In this mode the performance can slightly decrease, but the impact of the attacks is strongly reduced. The new operational mode responds to the threats described in Section 3.2 and it is generic enough to provide a unified response to them.

For the attacks that target the RWG mechanisms for discovery of new nodes and selection of custodians, the mitigation consists of ignoring cheap packets (ACK, OKTF, and BS) with "fake" identities. Of course, in the normal operation of the protocol none of the nodes have knowledge to distinguish good and fake identities. We propose that we have a chance of recognising such nodes if we add a low overhead mechanism to the protocol, namely creating a list of known nodes during the periods in which the mitigation is not enabled. This can be effectively done if a list is updated with identities of nodes that have sent REQF messages. This addition to the protocol is not wasteful of energy (given that transmission energy is the dominant factor) but uses up some storage at each node. We also expect a slight increase in the latency for detection of new nodes in the vicinity.

For the attacks that target the RWG mechanism for propagation of delivery status, the solution consists of going into a "suspicious mode". In this mode we restrict the update of the delivery information from the ACK packets received (i.e. do not set zeros to ones in the bit vector). More specifically, when the mitigation is enabled, the *informed* vectors of the messages contained in the node's local buffer are only updated from the *informed* vectors of the REQF, OKTF and BS packets. If an ACK is received the local *informed* vectors are just updated for the position that corresponds to the sender of the ACK, but the *informed* vector contained within the ACK packet is ignored. This mitigation imposes a heavier burden on the network resources. The information regarding the number of deliveries of each message is propagated slower than usual and the message is kept in the network for a longer time than needed increasing the transmission of packets around a 25%.

Obviously, the application of these techniques reduces the performance of the network if enabled indefinitely (we lose some of the strengths of RWG). This

is the reason why they are not an integrated part of the protocol specification. Instead, it is best to apply them just during an interval (I_m) after the detection of an attack. Further studies should show what are the optimal intervals to select for I_m in a given network environment.

5.4 Evaluation Metrics

Given the chaotic nature of the scenario we would not be able to use the classic detection rate (DR) and false positive rate (FPR) metrics for evaluation. This is due to the fact that the success of the approach is not measurable with those metrics neither on a per node basis nor on a network wide (average) basis. The locality of the attackers, the nature of the partitions, and the mobility of the nodes, all affect the results so that there are no meaningful homogeneous outcomes using these metrics. However, we will come back to them and analyse the above locality aspects in Section 5.6. Our main evaluation metrics for detection and mitigation are instead:

- **K-Delivery Rate (KDR):** Depending on the connectivity of the network, the message load, and the dynamics only a proportion of the messages sent are finally k-delivered. Thus, a good metric to evaluate the possible effects of an attack and its mitigation is the number of messages which are in fact k-delivered over the interval of study.
- **Packet Transmission Rate (PTR):** Another relevant metric is the number of packets transmitted during the interval of study. Besides being an indicator of the usage of bandwidth as a resource, the PTR is an indicator of the energy spent by the nodes, since the more transmissions the more energy is consumed.

5.5 Detection and Mitigation Results

The detection approach proposed in Section 4 has been tested with the two attacks and two combinations described in Section 5.2 (continuous and transient). In the following, whenever an attack is sensed the anomaly detector enables both mitigations at the same time (ignores ACK packets with possible bogus IDs, and does not update the informed vector on ACK packets received). The I_m interval is selected as 200 seconds.

Fig. 4 shows the effect of applying the detection and mitigation to the continuous draining attack. When the detection and mitigation mechanism is disabled the PTR in the network is around 90% higher than the normal rate as a result of the attack (except during the initial peak which is higher). However, when the mechanism is enabled, the PTR increases, but as soon as the attack becomes detected in most of the nodes, the mitigation actions are taken and the attack impact is reduced. Fig. 5 shows the transient draining attack, which as in the previous case increases the PTR with the same proportions. Nonetheless, in this case an initial peak of the PTR with mitigation is noticeable since the PTR in the simulation without attack is also increasing. In this case it is worth

mentioning that after the attack, the number of packets sent gradually returns to the normality as the mitigation is disabled within I_m of detecting in each node. In both cases the detection delay observed is about 10-30 seconds after the beginning of the attack for nodes close to the attackers.

Fig. 6 shows the effect of applying the detection and mitigation to the continuous grey hole attack. When the mechanism is disabled and the attack starts the KDR drastically drops to around 10% of the normal rate. With the mechanism enabled the KDR also drops, but not so low, and after a certain period it stabilises to values slightly below the values without an attack . In Fig. 7 the impact of the transient grey hole attack is shown, which as in the previous case drastically decreases the delivery ratio. The detection and mitigation responds similarly, but in this case it can be observed that the mechanism helps to a fast recovery once the attack has ended. With this attack the detection delay is longer and highly dependant on each node. The nodes close to the attackers show a detection delay around 10-60 seconds after the beginning of the attack for both continuous and transient modes. It is worth to say that this is a complex attack to mitigate since once the *informed* vector is sent there is a contagious impact on the other partitions while the mitigation is not enabled everywhere (since the detection is not strong enough in some places with the same threshold everywhere).

The results shown demonstrate that the approach is successful in creating a resistance to the attacks that conform to the given threat model despite the difficulties that the complexity of IC-MANET bring to the picture.

5.6 Locality and Classic Metrics

The most usual evaluation metrics for measuring the anomaly detection performance are the Detection Rate (DR)[1] and the False Positive Rate (FPR)[2]. In this section we show why these metrics are less meaningful in IC-MANET. We begin by discussing how to apply the metrics in the evaluation. To calculate these metrics we need to determine whether during the period of alarm the node was under attack. In intermittently connected networks the concept of being under attack for a particular node is not clear. Just considering a fixed attack interval for all the nodes is meaningless, since attacks do not always take place in a well-determined time interval and confined space. Nodes can be isolated from the attackers during some periods or can be too far from the attackers to be significantly affected by them. Besides, some attacks can be propagated further even if their source has stopped attacking, such as some types of flooding attacks. Hence, our attempt to account for the classic DR and FPR metrics, is based on tagging the packets sent by the attacker and the packets sent in response to them. Then, in each aggregation interval I_a(see Section 4.3) a node has been considered as being under attack if at least one of the tagged packets has been received in that interval.

[1] $DR = TP/(TP+FN)$ where TP stands for true positives and FN for false negatives.
[2] $FPR = FP/(FP + TN)$ where TN stands for true negatives.

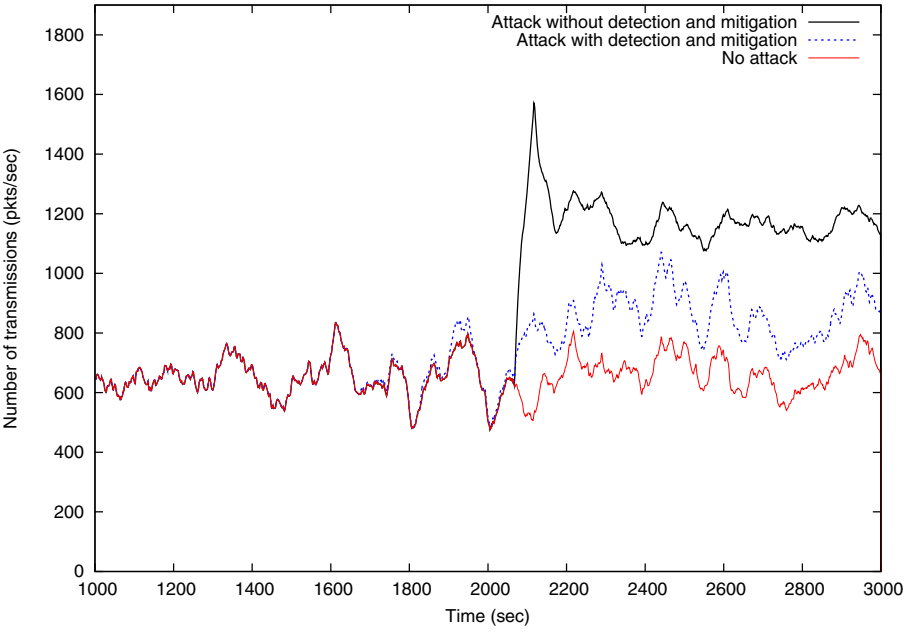


Fig. 4. Draining continuous attack

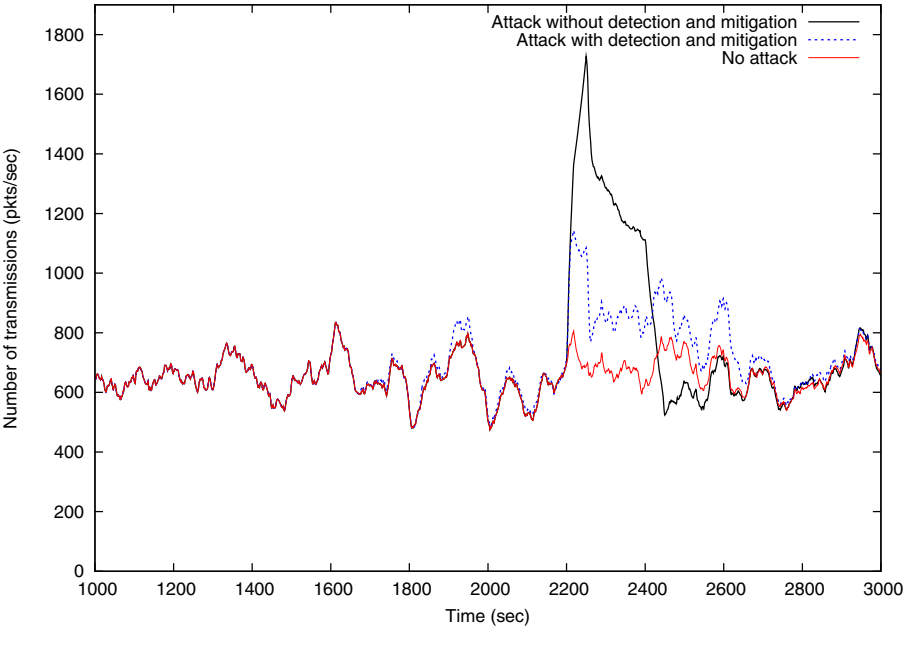


Fig. 5. Draining transient attack

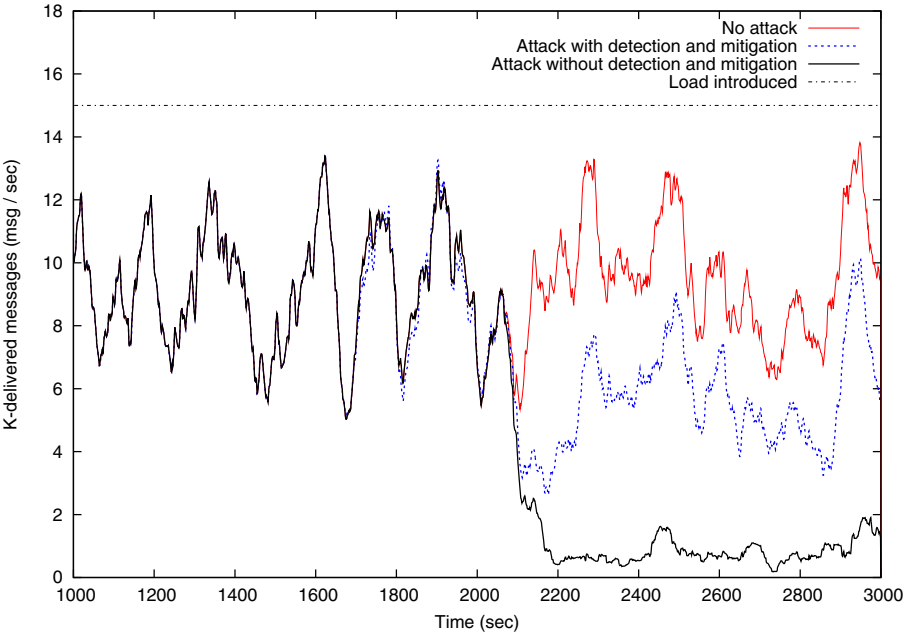


Fig. 6. Grey hole continuous attack

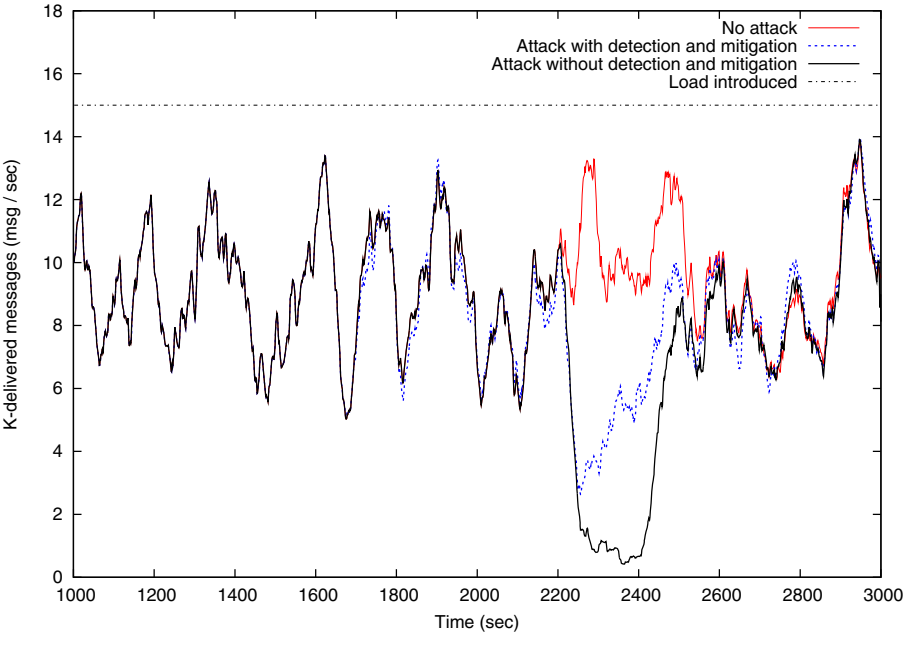


Fig. 7. Grey hole transient attack

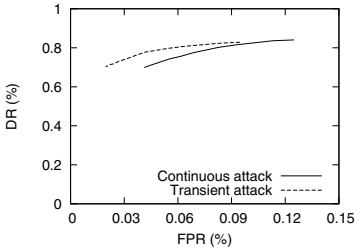


Fig. 8. ROC drain attack

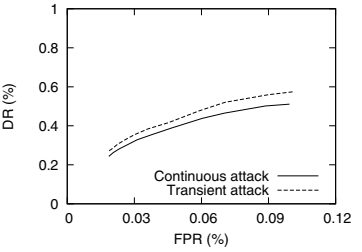


Fig. 9. ROC grey hole attack

The network wide average results obtained in terms of DR and FPR, by using different T_2 values have been depicted on Fig. 8 and 9. These numbers are computed by averaging the performance of all 25 anomaly detectors over the entire test interval. The curves demonstrate that in highly partitioned networks with very different conditions it is not feasible to analyse the results of the detection mechanism on an aggregate basis using these metrics. While the earlier results were convincing about the success of the approach, these curves show mediocre results overall.

We have observed that the traffic flow, the type of attack, and the number of attackers in each partition produce very different detection rates. The network topology in our disaster area is composed of eight partitions more or less stable along the whole simulation, with moving nodes acting as "bridges" over the partitions. Analysing the results node by node we have confirmed that the parameter with more influence over the detection performance is the proximity of the adversaries to the fair nodes. Table 1 show the best, worst, and average DR and FPR, for the continuous draining and grey hole attacks. Results in each column are categorised into different classes. Each class (different rows as described in column 1) shows the results aggregated for partitions that have similar number of adversaries, i.e. partions with no adversaries, partitions with 1 adversary, and so on. There are around 1/3 of the fair nodes in each class. The results, calculated with the alert aggregation threshold T_2 at 5%, demonstrate that the less the partition is affected by attacks the worse is the performance of the detection. That is, the classes with zero and one adversary are the ones that reduce the average detection performance. Note that despite having partitions with no adversaries, some attacks are received by sporadic contacts with other partitions.

Another aspect which has been observed is that in the transient cases the false positive rate is a bit lower than in the continuous cases. The reason is that the attacks are always detected with a small delay, but the alarm also persists when the attack is finished. Since the attack is not continuously received uniformly by all the nodes, because of their mobility, there are some gaps during which the alarms are enabled and counted as false positives. The continuous attacks are longer and present more of these gaps. This shows, once again, the complexity of the performance accounting using these metrics.

Table 1. Detection performance for the continuous attacks

# Adversaries per partition	Draining Attack						Grey Hole Attack					
	Average		Best		Worst		Average		Best		Worst	
	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR	DR	FPR
2	94%	6%	95%	6%	93%	8%	63%	8%	70%	5%	60%	10%
1	90%	5%	97%	3%	85%	7%	44%	4%	55%	2%	40%	7%
0	58%	5%	93%	4%	45%	8%	29%	6%	66%	3%	11%	9%

6 Conclusions

In this article we have presented a holistic anomaly detection and mitigation approach for dissemination protocols for intermittently connected networks. The approach has been integrated and evaluated in the Random Walk Gossip dissemination protocol applied within a disaster area scenario.

We have adopted a statistical-based detector algorithm to combat the typical resource constraints associated with the devices with respect to CPU power used for learning and detection. The threat model for which the approach has been validated focuses on making a big impact on fair nodes with little invested energy by the adversary. Moreover, the adversary behaviour is so similar to the normal behaviour that is hard to distinguish the attacks by creation of constraints, signatures or rules. So this environment is indeed a challenging environment.

Taking into account this threat model we have had to add a mitigation mode to the basic protocol operation. When in this mode, small modifications in the protocol create a chance of deciding when the own behaviour has to be changed due to a suspected attack. This is different from earlier works where identification of the culprit and individual isolation or specific treatment is the response. The integrated protocol can of course be run in the original no-mitigation mode when no attacks are expected and then no protection is provided either. Hence, the added detection-mitigation algorithm can be seen as an enhancement of an earlier protocol that works in a fair-play scenario. We believe this way of thinking can be generalised and applied in other dissemination protocols too.

Furthermore, our approach assumes full knowledge of the adversary about the protocol and even the anomaly detection scheme. The adversary cannot easily adapt to avoid detection by the algorithm due to the unpredictability of what learning has accomplished in the normality model. This is a simple and powerful aspect of our scheme.

The evaluation of the approach has demonstrated its effectiveness by showing resistance to the attacks using network performance metrics. In two attack modes, transient and continuous, we have shown that mitigation brings back the network to performance levels close to pre-attack scenarios. The analysis has also highlighted the complexity of using the classic metrics, detection rate and false positive rate, in highly partitioned networks. These metrics are not appropriate to measure the detection performance on a global basis in highly partitioned networks.

Future work includes identifying the applicability of the methods to more attack types, an intermittent version of the current attacks, and the addition of new threat models. It is also interesting to explore which parts of this resilience to attacks can be beneficially integrated into the dissemination algorithm. Current work includes the addition of two new components to the detection-mitigation loop. First, a diagnosis element that runs in parallel with a general (early) mitigation. This would be useful to adapting the mitigation without pinpointing attacker nodes. Second, an adaptive component that decides when and how to end a given mitigation phase, and a return to the less careful mode.

Another aspect in which more research is required is the study of impact of mitigation actions. When a node enables the mitigation, in some cases this may change the behaviour of the system and can be detected as an anomaly creating a recursive chain of alarms among the nodes. This is a complex problem because the behaviour of the system can be affected by the mitigation actions applied by all the nodes.

Acknowledgements

This work was supported by a grant from the Swedish Civil Contingencies Agency (MSB) and the national Graduate school in computer science (CUGS).

References

1. Denning, P.J.: Hastily formed networks. Communications of the ACM 49(4), 15–20 (2006)
2. Steckler, B., Bradford, B.L., Urrea, S.: Hastily formed networks for complex humanitarian disasters after action report and lessons learned from the naval postgraduate school's response to hurricane katrina. Technical Report, Naval Postgraduate School (2005)
3. Asplund, M., Nadjm-Tehrani, S.: A partition-tolerant manycast algorithm for disaster area networks. In: IEEE Symposium on Reliable Distributed Systems, pp. 156–165 (2009)
4. Aschenbruck, N., Gerhards-Padilla, E., Gerharz, M., Frank, M., Martini, P.: Modelling mobility in disaster area scenarios. In: MSWiM 2007: Proceedings of the 10th ACM Symposium on Modeling, Analysis, and Simulation of Wireless and Mobile Systems, pp. 4–12. ACM, New York (2007)
5. Ye, N., Chen, Q.: An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems. Quality and Reliability Engineering International 17(2), 105–112 (2001)
6. Yang, H., Luo, H., Ye, F., Lu, S., Zhang, L.: Security in mobile ad hoc networks: challenges and solutions. IEEE Wireless Communications 11(1), 38–47 (2004)
7. Prasithsangaree, P., Krishnamurthy, P.: On a framework for energy-efficient security protocols in wireless networks. Computer Communications 27(17), 1716–1729 (2004)
8. Farrell, S., Cahill, V.: Security considerations in space and delay tolerant networks. In: Second IEEE International Conference on Space Mission Challenges for Information Technology, Washington, DC, USA, pp. 29–38. IEEE, Los Alamitos (2006)

9. Liu, Y., Li, Y., Man, H., Jiang, W.: A hybrid data mining anomaly detection technique in ad hoc networks. International Journal of Wireless and Mobile Computing 2(1), 37–46 (2007)
10. García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G., Vázquez, E.: Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security 28(1-2), 18–28 (2009)
11. Nakayama, H., Kurosawa, S., Jamalipour, A., Nemoto, Y., Kato, N.: A dynamic anomaly detection scheme for AODV-based mobile ad hoc networks. IEEE Transactions on Vehicular Technology 58(5), 2471–2481 (2009)
12. Cabrera, J.B., Gutirrez, C., Mehra, R.K.: Ensemble methods for anomaly detection and distributed intrusion detection in mobile ad-hoc networks. Information Fusion 9(1), 96–119 (2008)
13. Chuah, M., Yang, P., Han, J.: A ferry-based intrusion detection scheme for sparsely connected ad hoc networks. In: Fourth Annual International Conference on Mobile and Ubiquitous Systems: Networking & Services, pp. 1–8. IEEE, Los Alamitos (2007)
14. Scalavino, E., Russello, G., Ball, R., Gowadia, V., Lupu, E.C.: An opportunistic authority evaluation scheme for data security in crisis management scenarios. In: ASIACCS 2010: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, pp. 157–168. ACM, New York (2010)
15. Thamilarasu, G., Balasubramanian, A., Mishra, S., Sridhar, R.: A cross-layer based intrusion detection approach for wireless ad hoc networks. In: IEEE International Conference on Mobile Adhoc and Sensor Systems Conference, pp. 854–861. IEEE, Los Alamitos (2005)
16. Sun, B., Wu, K., Pooch, U.W.: Zone-based intrusion detection for ad hoc networks. International Journal of Ad Hoc & Sensor Wireless Networks. Old City Publishing (2004)
17. Tseng, C.H., Wang, S.H., Ko, C., Levitt, K.: DEMEM: Distributed evidence-driven message exchange intrusion detection model for MANET. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 249–271. Springer, Heidelberg (2006)
18. Huang, Y.a., Lee, W.: A cooperative intrusion detection system for ad hoc networks. In: SASN 2003: Proceedings of the 1st ACM Workshop on Security of Ad Hoc and Sensor Networks, pp. 135–147. ACM, New York (2003)
19. Deodhar, A., Gujarathi, R.: A cluster based intrusion detection system for mobile ad hoc networks. Technical Report, Virginia Polytechnic Institute & State University
20. Wang, S.H., Tseng, C.H., Levitt, K., Bishop, M.: Cost-sensitive intrusion responses for mobile ad hoc networks. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 127–145. Springer, Heidelberg (2007)
21. Moore, D.S., Cabe, G.P.M.: Introduction to the practice of statistics, 5th edn. W. H. Freeman, New York (2005)

Community Epidemic Detection Using Time-Correlated Anomalies

Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken

Stanford University*
{oliner,ashutosh.kulkarni,aiken}@cs.stanford.edu

Abstract. An *epidemic* is malicious code running on a subset of a *community*, a homogeneous set of instances of an application. Syzygy is an epidemic detection framework that looks for time-correlated *anomalies*, i.e., divergence from a model of dynamic behavior. We show mathematically and experimentally that, by leveraging the statistical properties of a large community, Syzygy is able to detect epidemics even under adverse conditions, such as when an exploit employs both mimicry and polymorphism. This work provides a mathematical basis for Syzygy, describes our particular implementation, and tests the approach with a variety of exploits and on commodity server and desktop applications to demonstrate its effectiveness.

Keywords: epidemic detection, anomalies, community.

1 Introduction

Consider a set of instances of an application, which we call a *community*. Two examples of communities are all the mail servers in an organization or all the browsers on a cluster of workstations. Assume some subset of these instances, or *clients*, are compromised and are running malicious code. The initial breach (or breaches) went undetected and the existence of the exploit is unknown, so the malicious code may continue running indefinitely, perhaps quietly stealing computing resources (as in a zombie network), spoofing content, denying service, etc. We present a method for detecting such situations by using properties of the aggregate behavior of the community to reliably identify when a subset of the community is not behaving properly.

A client is either *healthy* and exhibits correct behavior or *infected* and exhibits incorrect behavior; our method detects *epidemics*, meaning when a subset of the community is infected. The user specifies what constitutes correct operation for individual clients by providing a *model*, which may be incomplete (omit correct behaviors), or unsound (admit incorrect behaviors), or both. For example, a community of web servers may be modeled by the typical distribution of response times each provides. The class of attacks we want to detect are those that cause

* This work was supported in part by NSF grants CCF-0915766 and CNS-050955, and by the DOE High-Performance Computer Science Fellowship.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 360–381, 2010.

undesirable deviation from normal behavior, regardless of the attack vector (e.g., buffer overrun, insider attack, or hardware tampering). Our focus is on detecting epidemics in a community composed of instances of a specific application, rather than the entire system or individual clients in the community, and this distinction leads to a different approach.

We describe an implementation of an epidemic detector, called Syzygy, that applies two main insights: (i) even if a single noisy model cannot reliably judge the health of a client, we can reduce the noise by averaging the judgements of many independent models and (ii) epidemics exhibit time-correlated behavior that is impossible to detect on a single client. Our method effectively leverages the statistical properties of a large community to turn noisy models into reliable community detectors and uses the temporal properties of an epidemic as a means for better detecting it.

Syzygy monitors each client's behavior and reports *anomaly scores*, which quantify the divergence of recent behavior from the model. For example, a client whose recent response times are unusually high may report a score that is above average (anomalous). Syzygy then computes the numerical average of all clients' scores and checks whether this *community score* exceeds a threshold. By doing these computations properly (see Section 3), we can make strong theoretical guarantees about our ability to overcome model noise and detect epidemics. Intuitively, we expect anomalies on individual clients in a large community to be common, but we do not expect anomaly scores from multiple clients to be strongly correlated in time, absent an epidemic.

We describe and analyze Syzygy's detection algorithm mathematically in Section 3. In our evaluation, we focus on the following questions:

—*Can Syzygy detect epidemics under realistic conditions?* In Section 4, we demonstrate that our method can leverage the community to detect a variety of epidemics in a cluster of commodity web servers even given noisy, incomplete client models. Syzygy does not require source code or specially compiled binaries.

—*How do client and community characteristics affect performance (i.e., false positives)?* In Section 5, we deploy Syzygy on the web browsers of a campus network and show that, despite very different client systems and user behaviors, healthy community behavior is a stable, reliable signal that is unlikely to generate excessive false positives (our deployments generated none). Indeed, as the community grows, Syzygy approaches a 100% detection rate with no false positives; given a sufficiently large training set and community, one can specify an acceptable false positive rate *a priori* and with high confidence. Even communities of only a dozen clients exhibit desirable properties. See Sections 3.3, 4.2, and 5.2–5.3.

—*What kinds of epidemics can Syzygy detect?* In Section 6, we conduct simulation experiments using commercial, off-the-shelf software and artificially powerful exploits (e.g., capable of nearly perfect mimicry) and demonstrate that the community enables Syzygy to detect epidemics under a variety of adverse conditions. Exploits may change their source code, perform different

malicious actions, or even use a different vector of infection across clients (see Section 3.2).

—*How good must client models be and how easy is it to acquire such models?* Syzygy works on top of existing client-based anomaly detectors, dampening noise and providing sensitivity to time-correlated behavior. Syzygy requires only that anomaly scores are mostly independent across healthy clients and higher, on average, for infected clients; the method is agnostic to what measurements are used to construct these scores.

Throughout the paper—using math, deployments, and simulations—we show that, in a large community, even simple, noisy models are sufficient for reliable epidemic detection. We conclude with a discussion of the issues involved with building a larger-scale deployment (Section 7). Many real security infrastructures are a constellation of tools; working in concert with other detection and response tools, and with low overhead and few practical requirements, Syzygy provides both new and more reliable information about epidemics.

2 Related Work

Syzygy detects malicious software running on clients in a community (epidemics) even under typical real-world constraints: the client model is incomplete, information about communication (network activity) is unavailable, and measurements are noisy. It may be impossible, given social engineering and insider attacks, to prevent all security breaches; a strength of Syzygy is that it can detect the bad behavior that follows a breach. In situations where the total damage is integral over time and the size of the infected community—such as when an exploit is stealing resources—the ability to detect such epidemics is crucial.

Anomaly-based intrusion detection has a long history [5, 27, 28, 29, 31, 35]. A commonly held view is that anomaly detection is fundamentally limited by the mediocre quality of the models that can be obtained in practice and therefore must necessarily generate excessive false positives in realistic settings (see, e.g., [2]). We agree with the gist of this argument for single clients, but we show in this paper that an appropriate use of a community can make strong guarantees even with noisy models.

Crucial, however, is how the community is used. Most previous systems that use a community at all use it only to correlate alarms generated locally on each client—the difficulty is that the alarm/no alarm decision is still made on the basis of a single client. Alert-correlation systems then try to suppress the resulting false alarms by correlating alarms from other clients or different detectors [4, 13, 36]. Other collaborative detection efforts that raise alarms only on individual clients include heterogeneous network overlays [44] and network anomaly detectors, such as by using cumulative triggers [15, 16] or alarm aggregation and correlation [1, 17, 32, 41]. Some work also uses correlation to characterize attack scenarios and causal flow [19, 26, 34].

Syzygy is fundamentally different from all of these systems in that it uses the aggregate behavior of the community to decide whether to raise an alarm for the community, not individual clients. The ability to make alert decisions based on analyzing the combined behavior of multiple clients is what gives Syzygy strong theoretical and practical properties that are absent from all previous work. There is prior work for file systems [43] and peer-to-peer networks [22, 23] that generate alerts based on aggregate behavior, but these do so without utilizing the statistical benefits of a large community.

Another category of work uses the community simply to gather data more quickly or to spread the burden of monitoring among many clients. For example, the Application Communities project [21] uses the community to distribute work; everything could be done on a single client, given more time. Syzygy uses the community in both these ways, as well; in contrast, however, it also looks for time-correlated deviations from normal behavior, which is not possible on a single client.

Syzygy was originally a detection component of the VERNIER security architecture [20]. Syzygy's role is to monitor instances of a target application for signs of infection: attacks on the security infrastructure or other applications within the client system, problem diagnosis, and reaction to the intrusion are all the responsibility of other VERNIER components. Among the various VERNIER detectors, Syzygy is specifically looking for time-correlated activity, as might be expected from a propagating worm or a coordinated attack. This specialization allows Syzygy to be small, lightweight, and asymptotically ideal while using the community in a novel way.

There are also uses of the community for tasks other than detection, such as diagnosing problems by discovering root causes [39] and preventing known exploits (e.g., sharing antibodies) [2, 3, 25]. Although other parts of VERNIER employ such measures, our focus is on detection.

3 Syzygy

Consider a community of n clients in which we wish to detect epidemics. During training, Syzygy observes the normal operation of the clients and builds a *model* (see Section 3.1). It is important to note that the specific choice of model is independent from the rest of Syzygy's operation; the only requirement is that the model produces an *anomaly signal* according to the constraints in Section 3.2.

While subsequently in monitoring mode, Syzygy periodically collects the most recent value of the anomaly signal (the *anomaly score*) from each client and checks whether the community's average anomaly score exceeds a threshold V. If so, Syzygy reports an *epidemic*. The properties of the anomaly signal are such that, given a large community, Syzygy can compute the threshold automatically at runtime and is insensitive to minor variations in this parameter. We explain these properties mathematically in Section 3.3 and support them experimentally in Sections 5.2 and 6.3.

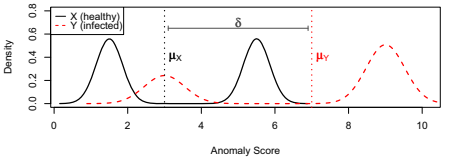


Fig. 1. An illustration of anomaly signals. Neither X nor Y are normally distributed, but $\mu_Y > \mu_X$, as required. The exploit may sometimes look "normal".

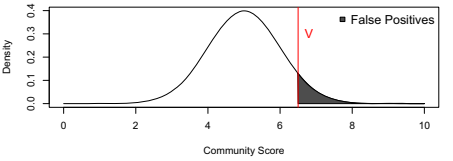


Fig. 2. A distribution of healthy community scores using hypothetical data. The threshold V determines what fraction of scores result in false positives.

3.1 Model

When applying our method to detect epidemics in a community, the user selects an appropriate client *model*, which uses some combination of signals that can be measured on individual clients to quantify how surprising (anomalous) recent behavior is. We require only that the model generate anomaly scores that are mostly independent across healthy clients and that it quantify how surprising recent behavior is, compared with historical behavior or a theoretical baseline.

The model for a community of servers might characterize normal behavior according to performance (see an example using request response times in Section 4), while the model for a community of web browsers might use code execution paths (see examples using system calls in Sections 5 and 6). The example models used in this paper could easily be refined or replaced with alternatives to match the attacks we want to detect: call stack content [8], execution traces [10], call arguments [24], remote procedure calls [12], etc.

3.2 Anomaly Signal

The anomaly signal decouples the choice of model from the rest of the system; any model that satisfies the properties explained in this section may be used with Syzygy. Each client keeps the server apprised of the client's *anomaly score*, the current value of the client's *anomaly signal*. This score is a measure of how unusual recent behavior is compared to a model of client behavior: a higher score indicates more surprising behavior than a lower score. (This is sometimes called the IS statistic [18] or behavioral distance [11].)

The distribution of anomaly scores generated by a healthy client (X) must have a mean (μ_X) that is less than the mean (μ_Y) of the anomaly score distribution of an infected client (Y), so we require $\mu_Y > \mu_X + \delta$. The larger the δ, the better, though any positive δ will suffice. Figure 1 illustrates two valid anomaly signal distributions, where X and Y are random variables such that both have finite mean and finite, positive variance.

More generally, let the anomaly scores from healthy client i, denoted a_i, be distributed like X_i (written $a_i \sim X_i$) and let $a_i \sim Y_i$ when client i is infected. Assume, without loss of generality, that all clients have the same distribution, i.e., let $X_i \sim X$ and $Y_i \sim Y$. The distributions may be standardized to enforce this

assumption, because only the mean and variance are relevant to our asymptotic results. If infected behavior does not differ from normal behavior, then δ will be unacceptably small (even negative); this can be resolved by refining the model to include more relevant signals or adjusting the model to amplify surprising behaviors. In this paper, we use two simple models (see Sections 4.1 and 5.1) that share a similar anomaly score computation (see Section 4.1), and both provided sufficiently large δ values to detect a variety of exploits.

3.3 Epidemic Detection

The Syzygy server computes the average anomaly score among the active clients; this *community score* C represents the state of the community. If $C > V$, for a tunable threshold V, the server reports an epidemic. Consider a healthy community of n clients and let $a_i \sim X$. Then, by the Central Limit Theorem, as $n \to \infty$, the community scores are distributed normally with mean μ_X and variance $\frac{\sigma_X^2}{n}$:

$$C = \text{average}_i(a_i) = \frac{1}{n}\sum_i (X) \sim \text{Norm}(\mu_X, \frac{\sigma_X^2}{n}).$$

When $E(|X|^3) = \rho < \infty$, where $E()$ denotes expected value, convergence happens at a rate on the order of $\frac{1}{\sqrt{n}}$ (Berry-Esséen theorem). Concretely, let $C' = C - \mu_X$, and let F_n be the *cumulative distribution function* (cdf) of $\frac{C'\sqrt{n}}{\sigma_X}$ and Φ the standard normal cdf. Then there exists a constant $B > 0$ such that $\forall x, n,\ |F_n(x) - \Phi(x)| \leq \frac{B\rho}{\sigma_X^3\sqrt{n}}$.

Consider now when some number of clients $d \leq n$ of the community have been exploited. The community score, as $n, d \to \infty$, will be

$$C = \frac{1}{n}\left(\sum_{i=1}^{n-d} X + \sum_{i=1}^{d} Y\right) \sim \text{Norm}\left(\frac{(n-d)\mu_X + d\mu_Y}{n}, \frac{(n-d)\sigma_X^2 + d\sigma_Y^2}{n^2}\right).$$

The rate of convergence guarantees that we get this asymptotic behavior at relatively small values of n and d, and even when $d << n$; in Section 6 we support this fact experimentally.

The threshold V must be set given the community size (n) and given the mean (μ_X) and standard deviation (σ_X) of the healthy client anomaly scores, but without knowing the size (d) and distribution (μ_Y and σ_Y) of the infected population, because those are unknown at runtime. We can pick any positive V between σ_X^2/n and $(\sigma_X^2/n) + \delta$ and guarantee that there exist n and d that give an arbitrarily high probability of perfect detection (FP=FN=0). Without knowing δ, however, the best strategy is to pick the lowest value of V such that the false positive rate is acceptable. Using the following analysis, we can compute and adjust V at runtime based on known quantities and a specified false positive rate; we do this using data from real deployments in Sections 4.2 and 5.2.

The expected rate of false positives is the fraction of the community scores in a community with no infected clients that falls above V. (See Figure 2.) This is

Table 1. A reference table of the terminology used in this paper. Let E be the event that Syzygy reports an epidemic and let H be the event that the community is healthy.

Term	Meaning
n	The total number of active clients in the community.
d	The number of infected clients in the community.
W_i	The size of the recent window on client i. We use $W_i = 1000$ measurements.
T_i	The silence threshold on client i. If the application records no measurements for T_i seconds, Syzygy generates a hiaton; if a client reports no anomaly scores for $2T_i$ seconds, the server marks it inactive.
a_i	Anomaly score. The instantaneous value of the anomaly signal $A_i(t)$ on client i.
X, Y	The distributions of anomaly scores for healthy (X) and infected (Y) clients.
C	Community score: average of the most recent anomaly scores from active clients.
V	The epidemic threshold. If $C > V$, Syzygy reports an epidemic.
δ	Defined as $\mu_Y - \mu_X$. Intuitively, the average distance between anomaly scores generated by healthy versus infected clients. One kind of mimicry attack drives δ toward zero.
r	The rate of a rate-limited mimicry attack: the application appears healthy a fraction $1 - r$ of the time and infected a fraction r of the time.
TP	True positive rate or detection rate. $P(E\|\neg H)$.
TN	True negative rate. $P(\neg E\|H)$.
FP	False positive rate, or Type I classification error rate. $P(E\|H)$.
FN	False negative rate, or Type II classification error rate. $P(\neg E\|\neg H)$.
F1 Measure	A summary metric with precision and recall weighted equally: $\frac{2TP}{2TP+FP+FN}$.

precisely the value of the parametrized Q-function, the complement of the normal cdf: $Q(\alpha) \equiv \frac{1}{\sqrt{2\pi}} \int_\alpha^\infty e^{-\frac{x^2}{2}} dx$. Let $H \sim \text{Norm}\left(\mu_X, \frac{\sigma_X^2}{n}\right)$ be the distribution of community scores in a healthy community of size n. The probability that a randomly selected community score will be a false positive is $\text{FP} = P(C > V) = Q\left(\frac{(V-\mu_H)\sqrt{n}}{\sigma_H}\right)$. Table 1 lists the significant terms and metrics used in this paper.

This analysis relies on two modest assumptions. First, the parameters μ_X and σ_X must characterize the future distribution of anomaly scores. A model that is out-of-date or produced with biased training data, for example, may produce anomaly scores inconsistent with the expected distribution. In Section 6.4 we explore the impact of using on one system a model produced for a different one and in Section 5.2 we show that even relatively heterogeneous machines produce predictable community score distributions. It is straightforward to detect when observed behavior disagrees with expectation, and the solution is to retrain the model. Second, during normal operation, client anomaly scores should be mostly independent. In situations like a network-distributed software upgrade,

innocuous dependencies may cause correlated behavior (i.e., correlated behavior without a malicious cause, which is our definition of a false positive). Indeed, it is indistinguishable from an attack except that one change to the software is authorized and the other is not. Such false alarms are easily avoided by making information about authorized changes to monitored applications available to Syzygy. Other sources of accidentally correlated behavior are quite rare; we observed no false alarms at all in a deployment with real users (see Section 5).

4 Detection Experiments

We first test Syzygy's ability to detect epidemics in a community using a cluster of 22 machines running unmodified instances of the Apache web server. Each machine has four cores (two dual core AMD Opteron 265 processors), 7 GB of main memory, and the Fedora Core 6 distribution of Linux. Each client serves streams of requests generated by a workload script. The workload generator, at exponentially distributed random times, makes requests from a list of 178 available HTML and PHP pages that includes several pages that do not exist and two pages for which the requester does not have read permission. We run the workload generator for 100,000 requests (∼2.8 hours) to train the model, then use those same training traces to set V so that we expect to get one false positive per week (see Section 3.3 for how we do this; also see Section 5.2 for more on false positives). We use Apache's existing logging mechanisms to record measurements (e.g., response times).

For this community, we aim to detect the following classes of attack: denial of service (DoS), resource exhaustion, content spoofing, and privilege escalation. Thus, we pick a client model that is likely to detect such attacks (see Section 4.1). We test Syzygy with two DoS attacks that prevent Apache from serving 1% or 10% of requests, at random, respectively; two resource exhaustion attacks that allow Apache to continue serving requests but gradually consume memory or CPU time, respectively; three content spoofing attacks that cause (i) PHP pages to be served in place of previously non-existent pages, (ii) PHP pages to be served in the place of certain HTML pages, or (iii) HTML pages to be served in place of certain PHP pages; and a privilege escalation attack that makes all page accesses authorized (no 403 Errors). We find that Syzygy can achieve high detection rates for these attacks with no false positives (see Section 4.2).

The clients in these experiments are homogeneous; in Section 5, we explore the effects of heterogenous hardware and varying user behavior with a deployment using an interactive application (the Firefox web browser). Section 6 contains additional experiments, in a more controlled environment, that explore the properties of much larger communities (thousands of clients) and more advanced exploits (capable of various degrees of mimicry).

4.1 Model

Assume that our security goal for this community is to ensure that clients are serving requests according to expected performance; that is, the request response

behavior should be consistent over time. During training, the model computes a frequency distribution of request response times and the maximum observed time between consecutive requests. This is just one choice of model and is not intrinsic to Syzygy.

When a request is made of the server, the model increments the counter associated with the response time s in a table indexed by response times (10 μsecond precision). From this frequency distribution, we compute a density function S_i by dividing each entry by the total number of observed response times. Thus, $S_i(s)$ is the fraction of times that response time s was observed on client i.

To incorporate timing in the model, which can help identify the absence of normal behavior (such as during a denial of service attack), we record the time between the start of each consecutive pair of requests. The model measures these times only when the application is *active*. A client is active when it reports its first anomaly score and becomes *inactive* after reporting an anomaly score accompanied by the `END` message. (See below for when this token is generated.) From these data, we set a *silence threshold* T_i for each client i, which we initially pick to be the maximum time between any two consecutive requests.

Monitoring. On the client, Syzygy monitors all requests made to the application. In addition, Syzygy may inject two kinds of artificial measurements into the sequence. The first, called `END`, indicates that the application has terminated (switched to *inactive*); Syzygy generates an `END` token when the application exits cleanly, terminates abruptly such as due to an error, or when the Syzygy client is closed cleanly. If an active client stops reporting scores for longer than the *timeout threshold*, currently set to $2T_i$ seconds, then the Syzygy server marks that client inactive without fabricating a token. The second artificial measurement, a *hiaton* [37] denoted `X`, indicates that no measurements were generated for longer than T_i seconds, including any `X`s produced via this process. In other words, at the start of each request, a timer starts; when this timer exceeds T_i, Syzygy generates a hiaton and resets the timer.

Each client maintains a window of the most recent W_i request response times, including the fabricated hiatons and `END` tokens. From this window, we compute the density function R_i, analogous to S_i, above. Thus, $R_i(s)$ is the fraction of times measurement s appears in the previous W_i measurements on client i.

Anomaly Signal. Let a_i be the most recent anomaly score and W_i be the size of the recent window for client i. The units of a_i and W_i may depend on the particular choice of model, but should be consistent across clients. In this paper, we measure the anomaly signal in bits and the window size in number of measurements. Our implementation computes a_i using Kullback-Liebler (KL) divergence with a base-2 logarithm. Roughly, this measures the information gained by seeing the recent window, having already observed the historical behavior. Specifically, over the measurements s in the density function for the recent window ($s \in R_i$), we have $a_i = \sum_s R_i(s) \log \frac{R_i(s)}{S_i(s)}$.

This computation can be updated incrementally in constant time as one measurement leaves the recent window and another enters it. To prevent division by

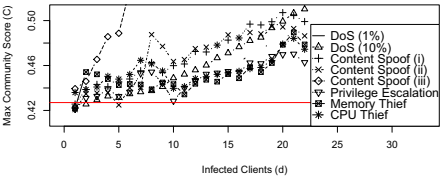


Fig. 3. Syzygy detected all of the attacks once the infection size was sufficiently large. The horizontal line is the epidemic threshold V.

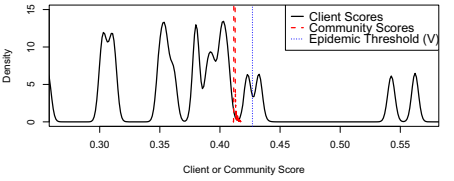


Fig. 4. Our client model is incomplete and noisy; anomalous behavior is common. The community scores, however, are extremely steady.

zero, the measurements in the recent window are included in the distribution S_i. By default, each client reports this score whenever there is new information available to the model (e.g., a request or hiaton), but it is straightforward to add feedback or batching to the client-server protocol to curb communication traffic (we do so in Section 5.3).

4.2 Results

Figure 3 shows the results of our detection experiments; there were no false positives in these experiments and detection latency was never more than a couple of seconds. Although some attacks are difficult to detect when only a few machines are infected (low d), Syzygy is able to correctly detect each attack once a sufficiently large number of clients are infected. In the case of the third (iii) content spoof attack, the behavior is anomalous enough on even a single client for our simple response time model to detect it; this is not true for most of the other attacks, meaning the community was crucial.

We achieved these high detection rates despite the fact that our behavior model was incomplete and noisy. Figure 4 shows part of the distribution of anomaly scores reported by individual healthy clients. In fact, these values ranged as high as 0.8 but we have truncated the graph for readability. In contrast, however, note that the healthy community scores stayed within a very small range (the dashed red line is actually a very slim Gaussian). The epidemic threshold V is the dotted line to the right of the cluster of community scores. Because the community scores are such a stable signal, they enable Syzygy both to reliably provide a low false positive rate and to be sensitive to minor—but not isolated—changes in client behavior.

In the subsequent sections, we discuss the benefits of distributed training, the effects of heterogenous hardware and user behavior, performance and overhead on a real network deployment, predicting and setting the false positive rate, performance in communities with thousands of clients, and Syzygy's robustness against tainted training data and advanced exploit behavior (like mimicry).

5 Deployment Experiments

For practical use, our method assumes that (i) a real deployment can scale to large numbers of clients across a realistic network topology and (ii) despite minor client variations, such as hardware and configuration differences, healthy anomaly score distributions are similar across clients. We verify that these assumptions hold in practice by deploying Syzygy on several dozen Linux workstations on a university campus. Most of these machines were 3.0 GHz Intel Core2 Duos with 2 GB RAM and the CentOS 5 operating system; exceptions include two laptops and (briefly) the Syzygy server, itself. Syzygy monitored the Firefox web browser via `strace` on Linux. Over the course of these two weeks of experiments, Syzygy reported no false positives.

5.1 Model

In the next two sections, we use a model of client behavior (different from Section 4) that uses short sequences of a program's system calls. This information can be gathered with low overhead and has been shown to be useful [9, 14]. We use sequences of six system calls to be consistent with previous work [7, 14, 22], but instead of using one of the existing *stide* or *t-stide* algorithms [33], the model uses an information theoretic approach with several additional modifications. During training, Syzygy computes a frequency distribution of system call sequences of length six and the maximum observed time between consecutive system call invocations. The computations are extremely similar to Section 4.1, but use system call sequences as measurements, instead of request response times.

Whenever a system call is invoked, the model concatenates the name of the call onto a sequence consisting of the previous five and increments the counter associated with that sequence. For example, on Mac OS X, while executing the command `echo hi`, we generate the following period-delimited sequence:

$s =$ `sigaction.writev.read.select.select.exit`.

Even when idle, many applications will continue to invoke system calls (e.g., polling for new work or user input). This behavior acts as a kind of heartbeat for the program, and its absence indicates unusual behavior just as much as the presence of, say, unusual system call sequences. For example, during one such execution of `echo hi`, the maximum time between system call invocations, according to `dtrace`, was 375 μs.

Using this kind of information about call sequences and timing, we construct a model analogous to the one for request response times in Section 4.1. The only differences are that the tables used to construct S_i and R_i are indexed by sequences and the recent window W_i has units of sequences. The anomaly signal is computed as described in Section 4.1.

5.2 Distributed Training

Over a period of roughly two weeks, we collected normal usage traces from 35 active clients. During the day, a median of 8 clients were active at a time. The

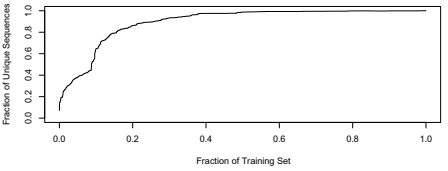


Fig. 5. Distributed training happens quickly: 25% of the data exhibits 90% of the unique sequences. Retraining a model (e.g., after a software upgrade) is efficient

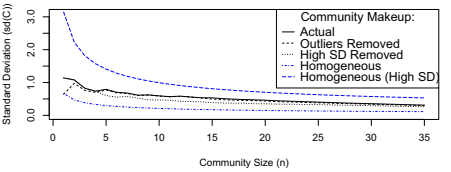


Fig. 6. Community scores converge in real data; variance comes from client variance, not system configuration or workload heterogeneity

first week of these traces is our training data and contains more than 2.2 billion sequences, of which approximately 180,000 are unique. As shown in Figure 5, most of the sequences were seen quickly (90% within the first 25% of the trace). The fact that training speeds up with community size is consistent with previous work [21]; Syzygy's distinctive use of the community occurs during the monitoring phase (Section 5.3).

During this training period, while the clients were reporting both the complete sequences and timestamps at an average of 100 KB/s, the average bandwidth usage at the server was 1160 KB/s (the peak was 3240 KB/s). The clients required less than 1% CPU each for the `strace` process and Syzygy script. With all 35 clients active, the server-side script was using 13% of the processor, on average, with peaks as high as 32%.

Even though the training data includes machines that are unlike most of the cluster, such as two laptops, we still find that the distribution of community anomaly scores within the training community converges toward a tight normal distribution. Figure 6 shows the standard deviation of the community score for increasing numbers of clients; in the figure, the clients "join" the community in reverse order of average anomaly score (so $n = 1$ represents the client with the highest average anomaly score). To evaluate the impact of heterogeneity, we also plot four hypothetical communities: "Outliers Removed," where the two laptops and the Syzygy server were replaced with the client with the lowest standard deviation, "High SD Removed," where the five clients with the highest standard deviations were replaced with five clones of the machine with the lowest standard deviation, and "Homogeneous" and "Homogeneous (High SD)," which are communities of n clones of the client with the lowest average anomaly score and highest standard deviation, respectively. The results show that variance in the community score comes not from client heterogeneity (the client in "Homogeneous (High SD)" was a normal cluster machine) but from client variance. The results also show that a larger community can compensate for client variance.

Section 3.3 shows how to compute the threshold V, given a desired false positive rate and the training data; these analytical results correspond well with what we observe experimentally. Using the data from our deployment, Figure 7 plots the appropriate choice of V for a desired false positive rate (note the log scale) and community size (n). The units of the false positive rate, for this

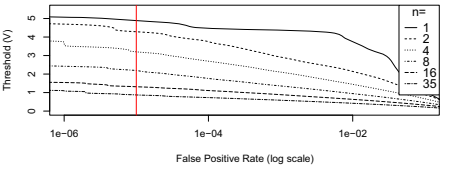


Fig. 7. For a given false positive rate and community size, we can compute the threshold V. The vertical red line, for instance, corresponds to about one false positive per six days.

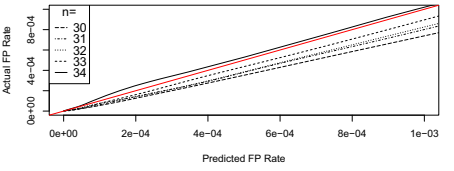


Fig. 8. The training data is a good predictor of the false positive rates seen in monitoring data. The threshold V can be set as high as necessary to achieve an acceptable rate of false positives.

deployment, are expected false positives per five seconds. The vertical line is a hypothetical target rate: 1×10^{-5} (about six days). The y-value at which this line intercepts each community size line is the threshold for that value of n.

5.3 Distributed Monitoring

After training is complete, Syzygy switches to monitoring mode. For these experiments, we set $T_i = \infty$ to prevent hiatons from being introduced. (We omit the exploration of T_i values for space reasons.) Over the course of a week, we collected just under 10 billion anomaly scores from the community. Five clients seen during training were not heard from again, while four new ones appeared. There were no epidemics nor other coordinated events during the monitoring period; the machines are part of the campus computing infrastructure, so we could not obtain permission to stage an epidemic.

The `strace` process on the client requires an average of 1–2% CPU overhead, and the Syzygy client script requires another 2–3% to calculate the anomaly scores and send them to the server. The server-side Syzygy process uses less than 1% of the CPU for a single client; our experiments suggest a server could easily handle more than a hundred clients (see Section 7).

Syzygy can either send one packet per anomaly score or buffer some number before reporting them. At an average rate of 2000 system calls per second, sending one packet per call would be inefficient. Buffering 100 scores with a short timeout to ensure freshness, for example, reduces the bandwidth requirements to 20 packets per second at 1.5 KB per packet ($\sim$ 30 KB/s), including the overhead of transmitting timestamps along with the anomaly scores, which we did for experimental purposes. Communicating the scores alone would require less than half this bandwidth.

Section 3.3 notes that achieving the target false positive rate requires that μ_X and σ_X accurately describe the future distribution of anomaly scores. Figure 8 quantifies that statement using the deployment data collected while Syzygy was in monitoring mode (data not used to build the model). The diagonal red line indicates perfect agreement. Even at very low false positive rates and small

community sizes, the modeling data was sufficient to allow good prediction of the false positive rate on real monitoring data.

6 Controlled Experiments

In this section, we test Syzygy in a controlled environment under various adverse conditions, using trace data from commodity applications and exploits capable of sophisticated behaviors.

An *experiment* is a binary classification problem in which Syzygy is given a sequence of anomaly scores for n clients and must decide whether 0 of them are infected (healthy) or whether $d \geq 1$ of them have been exploited (infected). Thus, an *example* is a set of n score vectors of length W_i. Ideally, Syzygy should report an epidemic iff one or more of the score vectors was produced by an infected client. We use standard metrics to evaluate performance on this classification problem: false positive rate (FP), false negative rate (FN), true positive rate (TP), true negative rate (TN), and F1 Measure ($\frac{2TP}{2TP+FP+FN}$), which combines precision and recall, weighting each equally.

For example, say we are measuring Syzygy's performance on a community of size $n = 100$ and epidemic of size $d = 5$. We produce an example of an infected community as follows. Say that we have already constructed models for all n clients and have the associated system call traces. To construct each of the $n-d$ healthy score vectors, we pick a window from the application trace, uniformly at random, and compute the anomaly scores as described in Section 4.1. (The sample window determines R_i.) Using exploit traces, we construct d infected score vectors. Syzygy then takes the n vectors of anomaly scores and computes the elementwise averages. If $C > V$ for *any* element C of the resulting community score vector, then Syzygy classifies the example as infected; otherwise, it classifies it as healthy. Using data described in Section 6.1, we plot the community scores for a pair of examples in Figure 9; a healthy example is on the left and an infected example on the right. In other words, in the plot, the first 1000 scores are from a healthy community, while the next 1000 are from an infected community—Syzygy

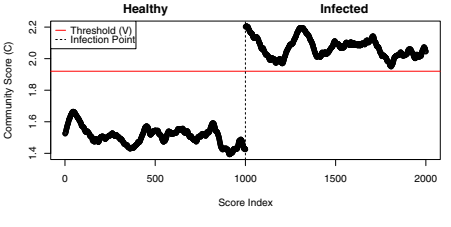


Fig. 9. A pair of examples, using Camino and the showpages exploit with $n = 100$ and $d = 5$, showing a TN and a TP

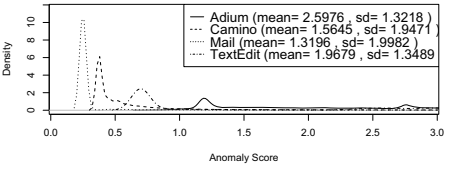


Fig. 10. Healthy anomaly distributions, plotted with a kernel density estimator. The bump at around 2.75 suggests Adium's model is imperfect.

Table 2. Training data. The Unique column indicates the number of unique length-six sequences. T_i is the maximum time from the beginning of one system call to the start of the next.

Application	**Version**	**Calls**	**Time** (sec)	**Rate** (calls/sec)	**Unique**	T_i (sec)
Adium	1.2.7	6,595,834	33,278	198.204	50,514	54.451
Camino	1.6.1Int-v2	113,341,557	57,385	1975.11	103,634	7.2605
Mail	3.3	106,774,240	48,630	2195.65	126,467	896.85
TextEdit	1.5 (244)	176,170	31,794	5.54098	4469	6031.4

classifies them based on V, reporting an epidemic when it sees the first score from the infected community.

We repeat this randomized process 1000 times per example to get statistically meaningful metrics. We always present Syzygy with an equal number of healthy and infected examples, though Syzygy does not use this fact in any way. This is not meant to reflect the base rate of intrusions in a system, but increases the precision of the metrics. As the size of the training set goes to infinity, it becomes irrelevant as to whether or not we remove the current trace file from the training set because its influence goes to zero. It is sufficient to select random windows from the traces because Syzygy is memoryless outside of each sample. Unless noted otherwise, we set $W_i = 1000$ sequences and $V = \mu_H + 2\sigma_H$, where H is the distribution of community scores for a community of size n, as in Section 3.3. We present the results of our controlled experiments in Sections 6.2–6.5.

6.1 Data

We collect system call and timing traces from commercial, off-the-shelf software under normal usage by the authors, using the utility `dtrace`. We use several desktop applications: a chat program (Adium), a web browser (Camino), a mail client (Mail), and a simple text editor (TextEdit). A summary of these data is provided in Table 2. When compared to the real deployments in Sections 4 and 5, we find that our simulations are a reasonable approximation. Note that, although Syzygy must build a dynamic model of application behavior, it does not need to learn exploit signatures.

Many exploits currently found in the wild are brazen about their misbehavior (large δ) and are therefore easy for Syzygy to detect (see Section 3.3). Instead, we focus in this section on Syzygy's distinguishing ability to detect next-generation exploits under adverse conditions. These exploits can infect the application at any execution point (i.e., multiple infection vectors), are privy to all of Syzygy's data and parameters, and can perform skillful mimicry. The adverse conditions include client heterogeneity and tainted training data.

In order to simulate such behavior, we use four next-generation exploits: *mailspam* infects Mail, then composes and sends a large number of emails (based on the open mail relay in the Sobig worm's trojan payload); *prompttext* infects TextEdit, then asks the user for input that it writes to a file (based on file creation and deletion seen in SirCam, Chernobyl, or Klez [40]); *screenshot* infects

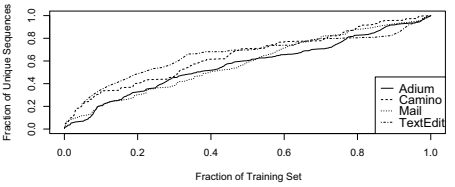


Fig. 11. The applications generate new sequences throughout training, with occasional bursts (e.g., program launches)

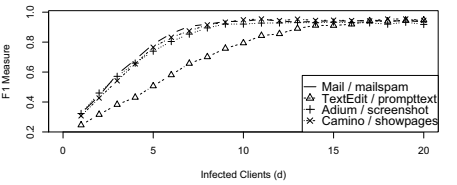


Fig. 12. F1 measure with $n = 100$ and varying infection size (d) using each of the four pairs of programs and exploits

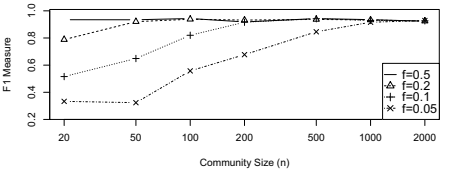


Fig. 13. F1 measure with varying community size and constant fraction $f = d/n$ infected, using TextEdit and prompttext

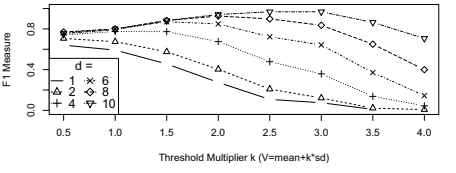


Fig. 14. F1 measure with $n = 100$ and varying threshold multiplier using traces from Mail and the mailspam exploit

Adium, then takes a snapshot of the current display (like prompttext but without user interaction); and *showpages* infects Camino, then loads a series of web pages (based on HTML proxies like Sobig's trojan, DoS payloads like Code Red and Yaha, and self-updating payloads like W32/sonic and W32/hybris [6]).

Except where noted, we gathered data using an Apple MacPro with two 2.66 GHz Dual-Core Intel Xeons and 6 GB of memory running Mac OS X 10.5.4, and the results we present are representative. Using the resulting model, we compute the distribution X of healthy client anomaly scores for each program (Figure 10). The results of Section 5.2 show that behavioral variance comes from client behavior over time, rather than client heterogeneity; the smartest way to gather a good data set was, therefore, to monitor a single client for a long time. Section 6.4 provides experiments supporting the merit of that decision.

We use the phrase "normal usage" to mean that no artificial workloads were generated nor were certain activities prescribed. As is evident from the rate of new sequences seen during training, plotted in Figure 11, we made no effort to train until convergence, nor to exercise the rarer features of these programs. We also do not separate sequences by thread, instead ordering them strictly by time of invocation. The resulting models are therefore small, imprecise, and incomplete, as we might expect to achieve in practice; the Syzygy performance numbers we present would only improve with better models.

6.2 Detection Performance

We first consider Syzygy's ability to detect epidemics for various sizes of community and infected population. Consider the experiments plotted in Figure 12

wherein a fixed-size community is being infected. Syzygy's performance improves with infection size, peaking, in this experiment, at around 10 exploited clients (10% of the community). Figure 13 shows, however, that with a sufficiently large community we require a vanishingly small fraction of the population to be sacrificed before we detect the exploit. Although the community and infected population are growing at the same rate, Syzygy's ability to detect the infection outpaces that growth.

6.3 Parameter Sensitivity

We next evaluate Syzygy's sensitivity to the threshold V. Figure 14 shows performance for various choices of V. Once the community and infected population are sufficiently large, we see the performance curve reach a maximum at a point between $V = \mu_X$ and μ_Y. Increasing the multiplier tends to increase precision, decrease recall, and decrease the false positive rate (which falls off like the tail of the normal distribution). To further visualize this, see Figure 15. As the number of clients grows, the normal and infected distributions become more clearly separated. This increasing noise margin suggests that the exact placement of the threshold does not strongly affect Syzygy's performance. Indeed, in the limit, all choices of threshold $\mu_X < V < \mu_Y$ yield perfect detection.

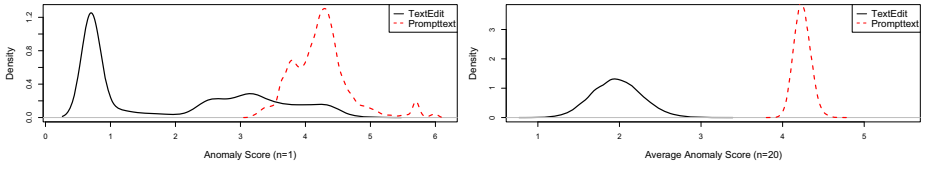


Fig. 15. The left plot shows anomaly signal density estimates for TextEdit and the prompttext exploit. There is no ideal position on the x-axis to set a threshold. On the right, we see that averaging scores across a number of clients yields a clearer separation.

6.4 Client Variation

We expect clients to differ in machine specifications and configurations, and for these to change over time. To test this situation, we ran the same applications as on our primary test machine (*System A*) on a second system (*System B*) with different specifications: an Apple PowerBook G4 with a single 1.33 GHz PowerPC processor and 1.25 GB of memory running Mac OS X 10.5.4. The data is summarized in Table 3. In Figure 16, we compare the anomaly scores for these Adium traces against those from the training system and the screenshot exploit. Although System B's average score is higher by Δ (its model is from another system), the programs behave similarly enough on both systems that unusual but healthy clients are not easily confused with exploits.

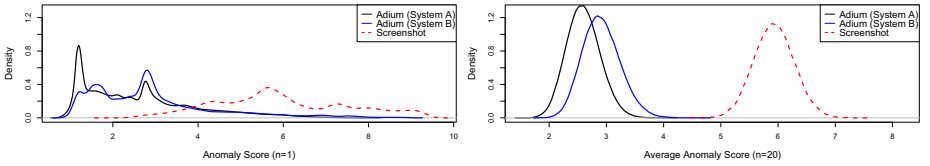


Fig. 16. Similar to Figure 15, except using the Adium program and giving data for both our primary system (System A) and the laptop (System B). All curves are based on the Adium model built using only System A.

Table 3. Data from OS X apps on a different client. The Unique column indicates the number of unique length-six sequences, and T_i is the maximum time from the beginning of one system call to the start of the next. The Δ column shows the empirically estimated average difference between anomaly scores on Systems A and B.

Program	**Version**	**Time** (sec)	**Rate** (calls/sec)	**Unique**	T_i (sec)	$\approx \Delta$
Adium	1.2.7	2093	54.8839	6749	47.457	0.31589
Camino	1.6.1Int-v2	3901	868.294	21,619	1.84077	0.60442
Mail	3.3 (926.1/926)	1126	16.2869	7963	421.645	0.53272
TextEdit	1.5 (244)	2506	92.8204	2925	528.164	1.17758

As the community grows, however, System B begins looking like an exploit. The healthy community score distribution variance, σ_H, shrinks, so V moves closer to μ_X, slowly passing below System B's average anomaly score. This contrived problem is easily remedied by using a model constructed from System B's behavior rather than System A's, or by normalizing the anomaly scores from System B as prescribed in Section 3.2. In practice, such a situation may arise when a client upgrades the application but does not retrain the model; if a client's anomaly signal remains high for long periods of time, this may indicate that the model is no longer valid—only when many clients make such changes would we expect spurious epidemic reports. Section 5 contains additional results related to client variation that suggest heterogeneity is not a problem in practice.

6.5 Mimicry and Tainting

An exploit can avoid detection if its behavior is sufficiently similar to the application's, from the perspective of a given model [38]. There are two ways an exploit might mimic application behavior: (i) by ensuring that the distribution of anomaly scores is sufficiently similar or (ii) by limiting the *rate* at which it exhibits bad behavior. Perfect mimicry, in which exploit behavior is indistinguishable from application behavior, can never be detected, by definition, using any behavior-based epidemic detector; however, we can show Syzygy is robust against a very high degree of mimicry and against rate-limiting an attack.

Scenario (i), mimicking the distribution, is quantified in Syzygy by the parameter δ. Recall that a lower value for δ means the two distributions are more

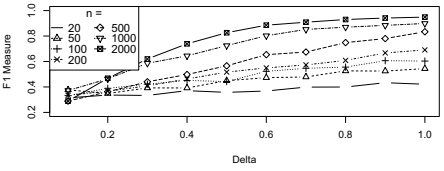


Fig. 17. Varying δ using Adium, with $d/n = 0.1$. Mimicry makes detection more difficult, but, at higher δs, performance improves logarithmically with n.

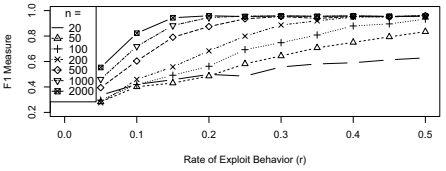


Fig. 18. Varying rate of bad behavior (r) using Camino and showpages, with $d/n = 0.1$. A sufficiently large community guarantees that bad behavior will overlap.

similar. Tainted training data is symmetric to mimicry: raising μ_X instead of lowering μ_Y. Either way, δ is decreased and the following results hold. Intuitively, these experiments simulate an exploit that makes system call sequences in similar (but not identical) proportions to the application. This is done computationally by generating anomaly scores from the application's distribution, then shifting them positively by δ. ($Y \sim X + \delta$.)

Figure 17 gives results from these experiments. Syzygy is able to detect fairly well even for low δ. The poor performance at the lowest δs, despite large communities, is almost exclusively a result of false negatives: V is set too high. With a lower V, we can get F1 > 0.6 even when $\delta = 0.1$, $n = 10$, and $d = 1$.

We now consider scenario (ii), limiting bad behavior to a fixed rate. Specifically, if the exploit spreads bad behavior out over time, in bursts that cumulatively account for a fraction r of the runtime per client, such that the community signal does not deviate above $\mu_X + V$, no epidemic will be reported. Mathematically, this attack corresponds to decreasing the effective infection size from d to dr. This, in itself, may be considered a victory under certain circumstances, such as when a worm may be contained so long as it does not spread too quickly [42]. In our experiment, we splice windows of infected anomaly scores into sequences of healthy anomaly scores, in proportions determined by the rate r. Figure 18 shows how Syzygy performs against this rate-limiting attack. Again, false negatives dominate the metric—with a better-chosen V, we can get F1 above 0.68 at $r = 0.05$ with as few as 10 clients.

7 Scalability

Mathematically, Syzygy's accuracy improves as the community grows, so it is crucial that the implementation scales well. This issue is independent of the analysis in Section 3. We described the infrastructure as using a central server, and demonstrated that it works for as many as 35 clients (Section 5). Communication is one-way (client to server) and there is no consensus or agreement protocol, so the total community traffic scales linearly with the number of clients.

This central server may be replaced, however, with alternatives that would increase scalability and avoid a single point of failure. One option is a server hierarchy; each server computes the community score for its children and reports

this value and the size of that sub-community to a parent server. This arrangement works precisely because the function used to compute the community score, `mean()`, is associative (when weighted by sub-community size).

In addition to communication overhead, there is monitoring overhead on the clients. This is typically a consequence of model choice and unaffected by community size. In our controlled experiments, the primary monitoring tool, `dtrace`, required less than 10% of one CPU even during heavy activity by the monitored application; the average usage was below 1%. In our deployment experiments with Firefox, Syzygy required less than 5% of the CPU on average, and 7% peak, including `strace` overhead (see Section 5.3). Using our `strace`-based implementation for Windows, however, the slowdown was noticeable. The overhead in our Apache deployment (see Section 4), which took advantage of the web server's built-in logging mechanism, was negligible. If overhead becomes problematic, then it may be worth changing the model to measure less costly signals. For example, Sharif et al [30] implemented control-flow monitoring with overhead comparable to our system call-based approach—this optimization would likely yield greater precision at lower overhead.

8 Contributions

Syzygy is an epidemic detection framework that looks for time-correlated anomalies in a homogeneous software community—precisely the behavior that would accompany an exploit as it executes among a set of clients. Our results show that Syzygy is effective at automated detection of epidemics, is practical to deploy, and scales well. Syzygy takes advantage of the statistical properties of large communities in a novel way, asymptotically approaching perfect detection.

Acknowledgments

The authors thank the members of the VERNIER team, especially Elizabeth Stinson, Patrick Lincoln, Steve Dawson, Linda Briesemeister, Jim Thornton, John Mitchell, and Peter Kwan. Thanks to Sebastian Gutierrez and Miles Davis for help deploying Syzygy, to Naeim Semsarilar for his invaluable contributions to the early stages of this work, and to Xuân Vũ for her input and support.

References

[1] Bouloutas, A., Calo, S., Finkel, A.: Alarm correlation and fault identification in communication networks. IEEE Transactions on Communications (1994)
[2] Brumley, D., Newsome, J., Song, D.: Sting: An end-to-end self-healing system for defending against internet worms. In: Malware Detection and Defense (2007)
[3] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-end containment of internet worms. In: SOSP (2005)
[4] Cuppens, F., Miege, A.: Alert correlation in a cooperative intrusion detection framework. In: IEEE Symposium on Security and Privacy, pp. 202–215 (2002)

[5] Debar, H., Becker, M., Siboni, D.: A neural network component for an intrusion detection system. In: IEEE Symposium on Security and Privacy (1992)
[6] Ellis, D.: Worm anatomy and model. In: WORM (2003)
[7] Eskin, E.: Anomaly detection over noisy data using learned probability distributions. In: ICML (2000)
[8] Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly detection using call stack information. In: IEEE Symposium on Security and Privacy (2003)
[9] Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A sense of self for unix processes. In: IEEE Symposium on Security and Privacy (1996)
[10] Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graphs for anomaly detection. In: CCS (2004)
[11] Gao, D., Reiter, M.K., Song, D.: Behavioral distance for intrusion detection. In: Zamboni, D., Krügel, C. (eds.) RAID 2006. LNCS, vol. 4219, pp. 19–40. Springer, Heidelberg (2006)
[12] Giffin, J.T., Jha, S., Miller, B.P.: Detecting manipulated remote call streams. In: USENIX Security, pp. 61–79 (2002)
[13] Gu, G., Cárdenas, A.A., Lee, W.: Principled reasoning and practical applications of alert fusion in intrusion detection systems. In: ASIACCS (2008)
[14] Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security 6(3), 151–180 (1998)
[15] Huang, L., Garofalakis, M., Joseph, A.D., Taft, N.: Communication-efficient tracking of distributed cumulative triggers. In: Intl. Conf. on Distributed Computing Systems (ICDCS) (June 2007)
[16] Huang, L., Nguyen, X.L., Garofalakis, M., Hellerstein, J., Jordan, M., Joseph, A., Taft, N.: Communication-efficient online detection of network-wide anomalies. In: IEEE INFOCOM (2007)
[17] Jakobson, G., Weissman, M.: Alarm correlation. IEEE Network (1993)
[18] Javitz, H.S., Valdes, A.: The SRI IDES statistical anomaly detector. In: IEEE Symposium on Security and Privacy (1991)
[19] King, S.T., Mao, Z.M., Lucchetti, D.G., Chen, P.M.: Constructing attack scenarios through correlation of intrusion alerts. In: CCS (2002)
[20] Lincoln, P., et al.: Virtualized Execution Realizing Network Infrastructures Enhancing Reliability (VERNIER), `http://www.sdl.sri.com/projects/vernier/`
[21] Locasto, M.E., Sidiroglou, S., Keromytis, A.D.: Software self-healing using collaborative application communities. In: NDSS (2005)
[22] Malan, D.J., Smith, M.D.: Host-based detection of worms through peer-to-peer cooperation. In: ACM Workshop on Rapid Malcode (2005)
[23] Malan, D.J., Smith, M.D.: Exploiting temporal consistency to reduce false positives in host-based, collaborative detection of worms. In: WORM (2006)
[24] Mutz, D., Valeur, F., Vigna, G., Kruegel, C.: Anomalous system call detection. In: TISSEC (2006)
[25] Newsome, J., Brumley, D., Song, D.: Vulnerability-specific execution filtering for exploit prevention on commodity software. In: NDSS (2006)
[26] Ning, P., Cui, Y., Reeves, D.S.: Constructing attack scenarios through correlation of intrusion alerts. In: CCS (2002)
[27] Paxson, V.: Bro: a system for detecting network intruders in real-time. Computer Networks 31 (1999)
[28] Porras, P.A., Neumann, P.G.: Emerald: event monitoring enabling responses to anomalous live disturbances. In: National Computer Security Conference, NIST/NCSC (1997)
[29] Sebring, M.M., Whitehurst, R.A.: Expert systems in intrusion detection: a case study. In: National Computer Security Conference (1988)

[30] Sharif, M., Singh, K., Giffin, J., Lee, W.: Understanding precision in host based intrusion detection. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 21–41. Springer, Heidelberg (2007)
[31] Smaha, S.: Haystack: an intrusion detection system. In: Aerospace Computer Security Applications Conference (1988)
[32] Staniford-chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagl, J., Levitt, K., Wee, C., Yip, R., Zerkle, D.: Grids—a graph based intrusion detection system for large networks. In: NIST/NCSC (1996)
[33] Tan, K.M.C., Maxion, R.A.: "Why 6?" Defining the operational limits of stide, an anomaly-based intrusion detector. In: IEEE Symposium on Security and Privacy (2002)
[34] Ullrich, J.: DShield—distributed intrusion detection system, `http://www.dshield.org`
[35] Vaccaro, H., Liepins, G.: Detection of anomalous computer session activity. In: IEEE Symposium on Security and Privacy (1989)
[36] Valdes, A., Skinner, K.: Probabilistic alert correlation. In: Lee, W., Mé, L., Wespi, A. (eds.) RAID 2001. LNCS, vol. 2212, p. 54. Springer, Heidelberg (2001)
[37] Wadge, W.W., Ashcroft, E.A.: Lucid, the dataflow programming language. A.P.I.C. Studies in Data Processing (1985)
[38] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: CCS (2002)
[39] Wang, H.J., Platt, J.C., Chen, Y., Zhang, R., Wang, Y.-M.: Automatic misconfiguration troubleshooting with PeerPressure. In: OSDI (2004)
[40] Weaver, N., Paxson, V., Staniford, S., Cunningham, R.: A taxonomy of computer worms. In: WORM (2003)
[41] Weaver, N., Staniford, S., Paxson, V.: Very fast containment of scanning worms. In: USENIX Security (2004)
[42] Williamson, M.M.: Throttling viruses: Restricting propagation to defeat malicious mobile code. In: ACSAC (2002)
[43] Xie, Y., Kim, H., O'Hallaron, D., Reiter, M., Zhang, H.: Seurat: a pointillist approach to anomaly detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 238–257. Springer, Heidelberg (2004)
[44] Yegneswaran, V., Barford, P., Jha, S.: Global intrusion detection in the DOMINO overlay system. In: NDSS (2004)

A Data-Centric Approach to Insider Attack Detection in Database Systems

Sunu Mathew[1,⋆], Michalis Petropoulos[2],
Hung Q. Ngo[2], and Shambhu Upadhyaya[2]

[1] Information Security,
Amazon.com Inc., Seattle WA 98104, USA
smathew@amazon.com
[2] Computer Science and Engineering,
University at Buffalo, Buffalo NY 14260, USA
{mpetropo,hungngo,shambhu}@buffalo.edu

Abstract. The insider threat against database management systems is a dangerous security problem. Authorized users may abuse legitimate privileges to masquerade as other users or to maliciously harvest data. We propose a new direction to address this problem. We model users' access patterns by profiling the *data points* that users access, in contrast to analyzing the *query expressions* in prior approaches. Our data-centric approach is based on the key observation that query syntax alone is a poor discriminator of user intent, which is much better rendered by *what* is accessed. We present a feature-extraction method to model users' access patterns. Statistical learning algorithms are trained and tested using data from a real Graduate Admission database. Experimental results indicate that the technique is very effective, accurate, and is promising in complementing existing database security solutions. Practical performance issues are also addressed.

1 Introduction

Ensuring the security and privacy of data assets is a crucial and very difficult problem in our modern networked world. Relational database management systems (RDBMS) are the fundamental means of data organization, storage and access in most organizations, services, and applications. Naturally, the ubiquity of RDBMSs led to the prevalence of security threats against these systems. An intruder from the outside, for example, may be able to gain unauthorized access to data by sending carefully crafted queries to a back-end database of a Web application. This class of so-called *SQL injection* attacks are well-known and well-documented, yet still very dangerous [1]. They can be mitigated by adopting suitable safeguards, for example, by adopting defensive programming techniques and by using *prepared* statements [2].

An *insider attack* against an RDBMS, however, is much more difficult to detect, and potentially much more dangerous [29,7,14]. According to the most

⋆ Work done as a graduate student at the University at Buffalo.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 382–401, 2010.

recent U.S. Secret Service/CERT/Microsoft E-Crime report, insider attacks constitute 34% of all surveyed attacks (outsiders constitute 37%, and the remaining 29% have unknown sources). For example, insiders to an organization such as (former) employees or system administrators might abuse their *already existing privileges* to conduct *masquerading*, *data harvesting*, or simply *sabotage* attacks [11].

More formally, the RAND workshop devoted to insider threats [8] defined an *insider* as "someone with access, privilege or knowledge of information systems and services," and the *insider threat* problem as "malevolent (or possibly inadvertent) actions by an already trusted person with access to sensitive information and information systems." Examples of insider attacks include *masquerading* and *privilege abuse* which are well-known threats in the financial, corporate and military domains; attackers may *abuse legitimate privileges* to conduct *snooping* or *data-harvesting* [29] with malicious intent (e.g., espionage).

1.1 Main Ideas

By definition, detecting insider attacks by specifying explicit rules or policies is a moot point: an insider is always defined *relative* to a set of policies. Consequently, we believe that the most effective method to deal with the insider threat problem is to statistically profile normal users' (computing) behaviors and raise a flag when a user deviates from his/her routine. Intuitively, a good statistical profiler should be able to detect non-stealthy sabotage attacks, quick data harvesting attacks, or masquerading attacks, because the computing footprints of those actions should be significantly different from day-to-day activities, from a statistical point of view.

The user profiling idea for insider threat detection in particular, and anomaly detection in general, is certainly not new (see, e.g., [30]). In the context of an RDBMS (or any problem requiring statistical profiling), the novelty is in the answers to two critical questions: (1) *what is a user profile (and how to construct it)?* and (2) *which machine-learning techniques and models should we adopt so that the profiles are* practically useful *for the detection problem?* By "useful" we mean some relevant classes of insider attacks can be detected to a good degree of accuracy. By "practical" we mean the method can be deployed and perform effectively in a real RDBMS. The novelty and contributions of this paper come from answering the above two questions.

Prior studies (e.g., [13,21,17,34,31,18]) have led to the development of intrusion detection systems (IDS) that aimed to protect databases from attacks. Our contribution is complementary, and is focused specifically on analyzing users' interactions with an RDBMS by means of database queries. Analysis of other behavioral features useful in insider threat detection (location of the attacker, informational correlation between consecutive queries, and temporal features such as time between queries, duration of session, etc.) is beyond the scope of this paper, and is considered future work.

Perhaps the most natural user "profile" is the set of SQL queries a user issues daily to the database, or more generally, some feature vectors representing past

queries. Indeed, [18] relied on the SQL-expression syntax of queries to construct user profiles. This approach has the advantage that the query processing of the insider detection system is computationally light: a new query is analyzed by some statistical engine; only queries accepted by the engine are then issued to the database. However, as we shall later demonstrate in this paper, this syntax-centric view is ineffective and error-prone for database anomaly detection in general, and for database insider threat detection, in particular. On the one hand, queries may differ widely in syntax yet produce the same "normal" (i.e., good) output, causing the syntax-based detection engine to generate false positives. On the other hand, syntactically similar queries may produce vastly different results, leading the syntax-based engine to generate false negatives.

Our main idea and also our conviction is that the best way to distinguish normal vs. abnormal (or good vs. malicious) access patterns is to look directly at *what* the user is trying to access – the result of the query itself – rather than *how* he expresses it, i.e. the SQL expressions. In other words, this data-centric approach values the *semantics* of the queries more than their *syntax*. When a malicious insider tries to acquire *new* knowledge about data points and their relationships, the data points accessed are necessarily different from the *old* (i.e., previously) accessed points. This deviation occurs in the data harvesting attacks as well as in the masquerading attacks (e.g., when an intruder gains access to an insider's account by means of a compromised account).

1.2 Contributions

Our first contribution is the proposed data-centric viewpoint, which to the best of our knowledge has not been studied in the database security and the insider threat literature. Intuitively, the data-centric approach has the following advantage: for an insider to evade our system, he has to generate queries producing results that are statistically similar to the ones he would have gotten *anyhow* with legitimate queries using his existing privileges, rendering the attempt at circumvention inconsequential. In contrast, in the syntax-based approach, queries with similar syntax can give different results: the attacker may be able to craft a "good-looking" malicious query bypassing the syntax-based detection engine to access data he's not supposed to access. This point is validated in Sections 3, 5 and 6.

The second contribution is a method to extract a feature vector from the result set of a query, which is the core of our answer to question (1) above. The dimension of the feature vector is only dependent on the database schema, but independent of the size of the database. In particular, the dimensionality of a query's feature vector is independent of how large the result set of the query is. This bounded dimensionality also partially addresses scalability and performance concerns the acute reader might have had. Section 4 details the method.

The third contribution is to address the following potential performance problem: a query has to be executed *before* the decision can be made on whether or not it is malicious. What if a malicious query asks for hundreds of gigabytes of data? Will the query have to be executed, and will our detection engine have to

process this huge "result set" before detecting the anomaly? These legitimate concerns are within the scope of question (2) above. We will show that this performance-accuracy tradeoff is not at all as bad as it seems at first glance. We experimentally show that a representative *constant* number of result tuples per query is sufficient for the detection engine to perform well, especially when the right statistical features and distance function (between normal and abnormal result sets) are chosen. Furthermore, these (constant number of) result tuples can be computed efficiently by leveraging the pipelined query execution model of commercial RDBMS's.

The fourth contribution, presented in Section 5, is a taxonomy of anomalous database access patterns, which is needed to systematically evaluate the accuracy of both the data-centric and the syntax-centric approaches.

The fifth contribution is a relatively extensive evaluation of several statistical learning algorithms using the data-centric approach. Specifically, for the masquerade detection problem on a real Graduate Admission data set, we found that k-means clustering works very well, with detection rates of around 95-99%. For detecting data harvesting, we develop an outlier detection method based on attribute deviation (a sort of clustering using the L_∞-norm) which performs well. Furthermore, this method is suitable when the features are only extracted from a constant number of tuples of the result set, thus making it practical.

In summary, though our results are derived in the limited context of insider threat detection with respect to database security, this paper is a first step in exploring the larger potential of the data-centric approach in anomaly detection.

Paper Outline. The rest of this paper is organized as follows. Section 2 surveys background and related work. Section 3 demonstrates the limitations of the syntax-based approach, thus motivating the data-centric approach introduced in Section 4. Section 5 gives a brief taxonomy of query anomalies facilitating the experiments presented in Section 6. We further discuss our solution, its implications, and future research directions in Section 7.

2 Related Work

IDSs with direct or indirect focus on databases have been presented in the literature [23,35]. In [22], temporal properties of data are utilized for intrusion detection in applications such as real-time stock trading. Anomaly detection schemes dealing with SQL injection attacks in Web applications were studied in [20,34]. SQL injection attacks are a specific kind of database query anomaly that is detected by our approach in a straightforward manner as we shall show.

Data correlation between transactions is used to aid anomaly detection in [17]. Similarly, in [32], dependency between database attributes is used to generate rules based on which malicious transactions are identified. The DEMIDS system [13] detects intrusions by building user profiles based on their working scopes which consist of feature/value pairs representing their activities. These features are typically based on syntactical analysis of the queries. A system to detect database attacks by comparison with a set of known legitimate database

transactions is the focus of [21]; this is another syntax-based system where SQL statements are summarized as regular expressions which are then considered to be "fingerprints" for legitimate transactions. Yet another syntax-based approach was considered in [27] for web databases, where fingerprints of all SQL statements that an application can generate are profiled. A binary vector with length equal to the number of fingerprints is used to build session profiles and aid in anomaly detection. This approach made assumptions such as restricting the number of distinct queries possible; these techniques may complement our approach in cases where the assumptions are valid. In [15], database transactions are represented by directed graphs describing execution paths (select, insert, delete etc.) and these are used for malicious data access detection. This approach cannot handle adhoc queries (as the authors themselves state) and works at the coarse-grained transaction level as opposed to the fine-grained query level. Database session identification is the focus of [36]: queries within a session are considered to be related to each other, and an information theoretic metric (entropy) is used to separate sessions; however, whole queries are used as the basic unit for n-gram-statistical modeling of sessions. A multiagent based approach to database intrusion detection is presented in [26]; relatively simple metrics such as access frequency, object requests and utilization, and execution denials/violations are used to audit user behavior.

Prior approaches in the literature that most resemble ours are [31] and [18]. The solution in [31] is similar in the use of statistical measurements; however the focus of the approach is mainly on detecting anomalies in database modification (e.g., *inserts*) rather than user queries. The query anomaly detection component is mentioned only in passing and only a limited set of features (e.g., session duration, number of tuples affected) are considered. The recent syntax-based work in [18] has the same overall detection goals as our work: detection of anomalies in database access by means of user queries. A primary focus on this paper will be on exposing the limitations of syntax based detection schemes; the approach in [18] will be used in this paper as a benchmark for evaluating the performance of our approach.

3 Limitations of Syntax-Centric Approach

This section demonstrates that two syntactically similar queries may generate vastly different results, and two syntactically distinct queries may give similar results. Consequently, SQL expressions are poor discriminators of users' intent. For example, a syntax-based approach may model a query with a frequency vector, each of whose coordinates counts the number of occurrences (or marks the presence) of some keywords or mathematical operators [18].

Consider the following query:

```
SELECT p.product_name, p.product_id
FROM PRODUCT p
WHERE p.cost = 100 AND p.weight > 80;
```

A syntactical analysis of this query and subsequent feature extraction (e.g., [18]) might result in the following features for query data representation – SQL Command – *SELECT*, Select Clause Relations – *PRODUCT*, Select Clause Attributes – *product_name, product_id*, Where Clause Relation – *PRODUCT*, Where Clause Attributes – *cost, weight*. Now consider the alternate query:

```
SELECT p.product_name, p.product_id
FROM PRODUCT p
WHERE p.cost > 100 AND p.weight = 80;
```

This query has the same syntax-based feature set as the previous one; however, the data tuples accessed in the two cases are vastly different.

Conversely, suppose we rewrite the first query as follows:

```
SELECT p.product_name, p.product_id
FROM PRODUCT p
WHERE p.cost = 100 AND p.weight > 80
AND p.product_name IS NOT NULL;
```

This query is syntactically different (three columns in the WHERE clause), but produces the same result tuples as the first (under the reasonable assumption that all products in the database have a valid product name). Most syntax-based anomaly detection schemes are likely to flag this query as anomalous with respect to the first.

Syntax analysis, even if very detailed (taking into account differences in the operand difference between '=' and '>' in the above examples) is complicated given the expressiveness of the SQL language, and involves determining *query equivalence*, which is difficult to perform correctly. In fact, query containment and equivalence is NP-complete for conjunctive queries and undecidable for queries involving negation [10]. Our data-centric approach bypasses the complexities and intricacies of syntax analysis.

4 Data-Centric User Profiles

A relational database often consists of multiple relations with attributes and relationships specified by multiple *primary key* and *foreign key* constraints. One can visualize a database as a single relation, called the *Universal Relation* [24], incorporating the attribute information from all the relations in the database.

Our approach profiles users as follows: for each query we compute a statistical "summary" of the query's result tuples. The summary for a query is represented by a vector of fixed dimension regardless of how large the query's result tuple set is. This way, past queries (i.e. normal queries) from a user can be intuitively thought of as a "cluster" in some high dimensional space. We'd like to emphasize that clustering is only one of several statistical learning technique we will adopt for this problem. The term clustering is used here to give the reader an intuitive sense of the model. When a new query arrives, if it "belongs" to the user's cluster, it will be classified as normal, and abnormal otherwise.

Table 1. Statistics Vector Format for Sample Database Schema

Database Schema		S-Vector Features
Relation	**Attribute**	
Product	Product.type(varchar)	Product.type.ncount Product.type.ndistinct
	Product.cost(numeric)	Product.cost.Min Product.cost.Max Product.cost.Mean Product.cost.StdDev Product.cost.Median

Our query summary vector is called an *S-Vector*. An S-Vector is a multivariate vector composed of real-valued features, each representing a statistical measurement; it is defined by the columns of the universal relation corresponding to a database. Each attribute of the universal relation contributes a number of features to the S-Vector according to the following rules.

Numeric Attributes: each numeric attribute contributes the measurements *Min* (value), *Max* (value), *Mean*, *Median* and *Standard deviation*.

Non-Numeric Attributes: the standard statistics do not make sense for non-numeric attributes (e.g., *char* and *varchar*). For categorical attributes, one option is to expand a k-value attribute into k binary-valued attributes (value 1 if the category is represented in the set of result tuples and 0 otherwise) and compute statistics on them as usual. However, the expansion of categorical attributes may result in an *S-vector* that has far too many dimensions, affecting the time-performance of the learner. We compromise by replacing each categorical attribute with two numeric dimensions representing the *total count* of values, as well as the number of *distinct values* for this attribute in the query result.

The S-Vector format for a database is determined by its schema; the value of the S-Vector for a query is determined by executing the query and computing the relevant attribute statistics based on the set of result tuples. Table 1 shows the S-Vector format for a database consisting of a single relation. To illustrate how an S-Vector value for a query is generated, consider the following query executed against the database in Table 1:

```
SELECT p.cost
FROM PRODUCT p
WHERE p.type = 'abc';
```

For this query, the result schema consists of the single column Product.cost and statistics computed on the result tuples are used to populate the PRODUCT.MIN, PRODUCT.MAX, PRODUCT.MEAN, PRODUCT.STDDEV and PRODUCT.MEDIAN features of the S-Vector format for the database – the result is the S-Vector representation of this query.

5 A Data-Centric Taxonomy of Query Anomalies

In order to evaluate the effectiveness and accuracy of a threat detection engine, a taxonomy of query anomalies can help us reason about potential solutions. Subsequent experiments can analyze the performance of detection schemes with respect to specific anomalies in light of this taxonomy. We shall classify query anomalies based on how "far" the anomalous query is from a normal query. From a data centric view point, two queries are represented by the two result sets, each of which consists of the result schema (the columns) and the result tuples (the rows). If the result schemas are (very) different, the two queries are different. If the result schemas are similar, then we need to look into how different the result tuples are. On this basis we classify query anomalies. Table 2 summarizes the taxonomy.

Table 2. A Taxonomy of Query Anomalies

Anomaly Cases	Types	Detected by Syntax-Centric?	Detected by Data-Centric?	Attack Models
Type 1 Different Schema/ Different Results		Yes	Yes	Masquerade
Type 2 Similar Schema/ Different Results	(a) Distinct Syntax	Yes	Yes	SQL-Injection Data-Harvesting
	(b) Similar Syntax	No	Yes	
Type 3 Similar Schema/ Similar Results	(a) Different Syntax/ Similar Semantics	False Positive	Yes (True Positive)	Data Harvesting
	(b) Different Syntax/ Different Semantics	Yes	No (Rare)	

Type 1: Distinct Schema and Tuples. Anomalous queries of this type have result sets whose columns *and* rows are very different from those of normal queries. Intuitively, anomalous queries of this type should be detected by both the syntax-based and the data-centric approaches. The syntax-based approach works because queries that differ in the result schema should have distinct SQL expressions (especially in the SELECT clause). The data-centric approach works because the S-vector of the anomalous query not only differ in the dimensions (the result schema) but also in the magnitudes in each dimension (the statistics of the result tuples). From the insider threat perspective, data harvesting and masquerading can both result in this type of anomaly. As an example, consider the following two queries to the database described in Table 1:

```
Query 1: SELECT p.cost                Query 2: SELECT p.type
         FROM PRODUCT p                        FROM PRODUCT p
         WHERE p.type = 'abc';                 WHERE p.cost < 100;
```

Distinguishing these kinds of queries has received the most attention in the literature (e.g., [18]) especially in the context of masquerade detection and Role Based Access Control (RBAC) [28], where different user roles are associated with different authorizations and privilege levels. An attempt by one user-role to execute a query associated with another role indicates anomalous behavior

and a possible attempt at masquerade. Syntax-based anomaly detection schemes have been shown to perform well for this case and we experimentally show later that data-centric schemes are also equally effective.

Type 2: Similar Schema, Distinct Tuples. Anomalous queries of this type have result sets whose columns are similar to normal queries, but whose rows are statistically different. The syntax of type-2 anomalous queries might be similar to or different from normal queries. For example, consider the following normal query:

```
SELECT *
FROM PRODUCT p
WHERE p.cost = 100;
```

Execution of this query results in the schema `(p.type, p.cost)` and data corresponding to the `WHERE` condition `p.cost = 100`. On the one hand, the following type-2 anomalous query has the same result schema as the normal one with a statistically different result tuple-set (matching the additional constraint of the product type):

```
SELECT *
FROM PRODUCT p
WHERE p.cost < 100 AND p.type = 'abc';
```

The SQL expression syntax is also distinctly different from the normal query. The WHERE clause has an additional attribute that is checked (`p.type`) compared to the previous query. On the other hand, the following type-2 anomalous query has the same result schema and also similar syntax as the normal query:

```
SELECT *
FROM PRODUCT p
WHERE p.cost < 100 AND p.cost > 100;
```

Yet the result tuples are the *complement* of that of the normal query. Thus, we further classify type-2 anomalous queries into type-2a and type-2b, where type-2a contains type-2 anomalous queries whose syntax are also distinct from normal ones, and type-2b contains the rest. The intuition is that a syntax-based scheme such as that in [18] is unlikely to be able to detect type-2b anomalous queries. Indeed, the scheme in [18] represents the above type-2b query variation and the original example query identically. Furthermore, type-2b anomalous queries can be rewritten in multiple ways (e.g. `p.cost != 100`), varying combinations of constants, arithmetic and logical operators; even very detailed syntax-based models may be hard-pressed to consider all variations. We will show that data-centric schemes are likely able to detect both of these anomalous types.

From the insider threat perspective, data harvesting and masquerading can both result in type-2 anomaly. Another example of a well-known attack class that may fall in this category is *SQL injection* since a typical attack is one that injects input causing condition checks to be bypassed resulting in the output of all tuples – e.g., a successful exploit of the first example above may lead to the execution of:

```
SELECT *
FROM PRODUCT p
WHERE 1;
```

Type 3: Similar Schema and Tuples. A query whose execution results in a similar schema and tuples as a normal one is considered to be similar from a data-centric viewpoint. Clearly, if the queries also have the same syntax, then their resulting schemas and tuples are the same and they *are* identical from both the data-centric and syntax-centric view.

The interesting case arises when a query producing the same result as a normal query is syntactically different from the normal query. The question is, should we consider such a query "anomalous"?

On the one hand, it seems to be obvious that a user accessing the same data schema and tuples as those in his normal access patterns should not be flagged as malicious regardless of how different the syntax of the queries he issued. For example, the following two queries should be considered identical:

```
SELECT p.type
FROM PRODUCT p
WHERE p.cost < 100;
```

```
SELECT p.type
FROM PRODUCT p
WHERE p.cost < 100 AND p.type IN (
   SELECT q.type
   FROM PRODUCT q
);
```

The two queries have identical outputs, semantics, and thus user intent. We will refer to an "anomalous" query of this type a *type-3a query*. Note again that "anomalous" is not the same as "malicious." Our approach will not raise a red flag, while the syntax-based approach would issue a false positive.

On the other hand, two queries resulting in the same output might actually reveal *more* information than what is in the output. To see this, we have to look a little deeper into the *semantics* of the queries. Consider the following query in relation to the ones from the previous paragraph.

```
SELECT p.type
FROM PRODUCT p
WHERE true;
```

Now assume, for the sake of illustration, that the attacker is attempting to see all product types (data harvesting). If the above query returns more (or different tuples) with respect to the first example, then the data-centric approach should, conceptually detect this. But in the rare case when the result tuples are exactly the same, this would (as expected) be permitted by the data-centric approach. However, the attacker has now gained the additional information (based on his results from the query from the previous paragraph), that all product types in the database cost less than 100, and has refined his knowledge regarding some entity. We call to this type of anomalous query *type-3b*.

This kind of successive knowledge accrual has received much interest in the areas of privacy preserving data mining and query auditing ([4,19]). The attack

here arises from information refinement through temporal interaction between a user and a database and not from a property of the query itself (i.e., its syntax or result data). Exploiting temporal features from a data-centric viewpoint is an important future research direction of ours. It should be noted, however, that it is difficult for an attacker to intentionally exploit this condition, since presumably he is unable to predict the nature of query output to ensure that result statistics are unchanged from a normal query. In any case, addressing this type of attacks is beyond the scope of this paper.

6 Experimental Validation

6.1 The Test Environment

The test environment consists of a real and currently active web application for Graduate Student Admissions (called *GradVote*) that relies on a Postgresql database at the back-end. Users of the system query the database primarily via the web application. The users fall into several roles, including *Chair*, *Faculty* and *Staff*.

The database schema consists of 20 relations with multiple (over 20 for some tables) numeric and non-numeric attributes and 39 multi-level views (i.e., the views refer to base relations as well as to other views). The training and testing datasets consist of tens of thousands user queries labeled both by individual username as well as by user-role. These views are significantly complex, possessing multiple subqueries, complex joins and statistical attributes.

Our system, *QStatProfiler*, is positioned in the middle of the interaction channel between the application and the database. It observes the queries to the database as well as the results returned to the application. As queries are submitted to the database and result tuples are returned, QStatProfiler simultaneously computes query statistics and the S-vectors for the queries. QStatProfiler is flexible and can accommodate a variety of machine learning/clustering algorithms. We shall elaborate on different algorithms and anomaly detection goals later.

Query Filtering: The first task of *QStatProfiler* is profiling users or roles. It is thus necessary to ignore queries that are are common for all users. For example, the application may issue a query to the database to obtain the currently active list of users, or the time-line for a particular activity, and so on. These queries may sometimes be generated as part of application startup. This set queries is well-known *a priori*, since they may be embedded in the application code and can be ignored while profiling. In our case, we maintain a list of *url* tags that indicate common application queries, called *Framework Queries* by *QStatProfiler*.

Query Parsing and Unfolding: This component is concerned with obtaining the mapping between the schema of the result set and the overall schema of the database. The syntax of a user query may not refer directly to elements of the base database schema (i.e., base relations and their attributes). References may be made to views that might refer to other views; the use of aliases and in-line subquery definitions can complicate the task of schema mapping. *QStatProfiler* uses a query parsing component that is tailored to the *Postgresql* SQL syntax.

Query parse trees are constructed and analyzed to determine the subset of the database relations and attributes that are present in the result tuples. The output of this phase is thus a set of relations and attributes that describe the result tuples, from which S-vectors are constructed.

6.2 Approximating S-Vectors

As alluded to earlier, having to execute a query before classifying it as anomalous is a legitimate performance concern – we address this issue in this section.

First, we argue that the approach does not impose significant *additional* burden to the database server. In most application environments (e.g., web database applications), execution of database queries is part of typical application function. For example, a user might submit queries through a web form; the queries are executed at a remote database server and the results are made available to the application. Our system operates as a passive component between the application and the database server, observing queries and the corresponding results without disrupting normal functioning. The database does not experience any additional load due to the anomaly detection system; the computational cost of calculating result statistics falls on a different host that runs the ID system (QStatProfiler).

Second, the data-centric approach needs to see some data, necessitating some performance penalty if we compare it to the syntax-centric approach on a malicious query that the syntax-centric approach is able to detect (a true positive!). However, as we shall see, the execution of one pipelined round in the RDBMS is sufficient for the data-centric engine to perform well. The extra burden put on the server is minimal, and is only marginally worse than the syntax-centric approach when that approach produces a true positive while ours produces a false negative (type-3b queries, e.g., which are difficult for attackers to construct). This marginal penalty is more than offset by queries on which our approach produces a true positive while the syntax-based approach gives a false negative (type-2b queries, e.g., which are easy for attackers to construct).

We propose to utilize only k tuples from the result set to build the corresponding S-vector. We tested two ways to choose k tuples from a result set.

Initial-k tuples: Only the initial k tuples in the result set are used to approximate the entire result set. Statistics computed from these tuples are used to generate the *S-Vector* representation of the query. As soon as the S-Vector is classified as anomalous, we can stop the rest of the pipelined rounds from the database, avoiding extra execution overheads.

Random–k tuples: k tuples are chosen at random from the complete result set – the S-vector of these k tuples are computed to represent the result set. This approach is expected to produce better accuracy as compared to the initial-k approach as it is not likely to be sensitive to specific orderings of the result tuples by the database (this is especially important if the SQL query contains 'ORDER BY' clauses). Fortunately, we show that our choice of the distance function seems to be *insensitive* to result set ordering, as long as the set is not too small.

Table 3. Detection Percentage (%) – Type 1 Anomalies (Role Masquerade)

Roles		Syntax-Centric			Data-Centric						
	Algorithm	**C quip.**	**M quip.**	**F quip.**	**S-V (all)**	**S-V I(20)**	**S-V R(20)**	**S-V I(10)**	**S-V R(10)**	**S-V I(5)**	**S-V R(5)**
Chair vs. Faculty	N-Bayes	81.67	85.33	75	85	85	82.67	78.33	77	81.67	90
	Dec. Tree	**88**	**87.67**	**87.67**	**96.33**	88.3	88.3	89	88.67	88.67	88.67
	SVM	83.3	81	**87.67**	82.33	74.67	77	71.33	75.67	68	74.33
	Clustering	73.3	72	65.67	92	**92.67**	**92.33**	**94**	**94**	**92.67**	**93.33**
Chair vs. Staff	N-Bayes	58	93.5	95.5	60.5	59	60.5	62	57.5	62.5	60.5
	Dec. Tree	75	**88**	**96**	**95.5**	92.5	**96**	96	93	95	92.5
	SVM	51.5	84.5	**96**	80	84	85.5	78.5	81.5	85.5	82
	Clustering	**88.5**	85.5	90.5	91.5	**99**	**96**	**98.5**	**95**	**100**	**96**
Faculty vs. Staff	N-Bayes	84.33	90.67	93	58.67	61.3	60.3	60.3	59.3	63	60
	Dec. Tree	**90**	**93.67**	**95.67**	89.3	92.3	91.67	92	93.67	91.33	91.67
	SVM	87	93	**95.67**	69.67	71.67	71	69.33	72	68.67	72
	Clustering	78.7	73.3	78	**99**	**100**	**99.6**	**99.3**	**99.3**	**100**	**99.3**

6.3 Detecting Type 1 and 2a Anomalies, and Masquerade Attacks

This section will show that the data-centric approach works slightly better than the syntax-centric approach for type 1 and type 2a anomalies. The fact that both approaches work well is to be expected by definition, because both the syntax and the query result statistics are different in type 1 and type 2a anomalies. The syntax-centric scheme in [18] has been shown to perform well in detecting role-based anomalies. Because the results are similar and due to space limitation, we will present only the type-1 anomaly results. Our experiments are also aimed at evaluating the accuracy of both approaches in detecting *role masquerade attacks*. Recall that each query for *GradVote* comes with a user role, and the execution of a typical query in one role by a user with a different role constitutes an anomaly.

Syntax-Centric Features: For the sake of completeness, we briefly summarize the syntax-centric data formats of [18]. Three representations are considered: *Crude* (C-quiplet), *Medium* (M-quiplet), and *Fine* (F-quiplet). *C-quiplet* is a *coarse-grained* representation consisting of the SQL-command, counts of projected relations, projected attributes, selected relations, and selected attributes. *M-quiplet* is a *medium-grained* format recording the SQL command, a binary vector of relations included in the projection clause, an integer vector denoting the number of projected attributes from each relation, a binary vector of relations included in the selection clause, and an integer vector counting the number of selected attributes from each relation. *F-quiplet* is fine-grained, differing from the M-quiplet in that instead of a count of attributes in each relation for the selection and projection clauses, a binary value is used to explicitly indicate the presence or absence of each attribute in a relation in the corresponding clauses.

Test Setup: The available dataset of queries is labeled by the roles *Staff*, *Faculty*, and *Chair*, in addition to *Framework*, for the common application-generated queries. The query set is randomized and separated into *Train* and *Test* sets of 1000 and 300 queries, respectively. Four query data representations are tested: our *S-Vector* (dimensionality 1638) and the syntax-centric *C-quiplet* (dimensionality 5), *M-quiplet* (dimensionality 73), and *F-quiplet* (dimensionality 1187).

Table 4. Detection Percentage (%) – Type 2b Anomalies (Data Harvesting Attacks)

Algorithm		Syntax-Centric			Data-Centric						
		C quiplet	M quiplet	F quiplet	S-V (all)	S-V I(20)	S-V R(20)	S-V I(10)	S-V R(10)	S-V I(5)	S-V R(5)
Cluster Outlier	Detection	23.5	26.4	17.64	**83.87**	12	67.7	6.4	45.1	6.4	35.4
	False Positive	14.47	11.84	15.8	**10.5**	3.9	6.5	3.9	5.2	2.6	6.6
Attrib Deviation	Detection	0	17.64	2.9	**87**	**87**	**87**	**87**	**87**	12.9	64.5
	False Positive	0	4.8	4.8	22.6	26	**15**	23.8	**15.8**	23.8	20.4

Four supervised learning algorithms are tested with each of these feature sets: Naive Bayes, Decision Tree Classifier, Support Vector Machines, and Euclidean k-means clustering (see, e.g., [6]).

The results for the binary classifiers for masquerade detection are shown in Table 3 (the best performance for each format with respect to separating user roles is shown in boldface). In the table, I(k) and R(k) denote the Initial-k and Random-k S-Vector approximations. There are two main results. *First*, the performance of the S-Vector based detection using k-mean clustering is virtually uniformly better than the syntax-based schemes. In many cases, the detection rates are approaching 100%. Note also that, the *false positive* rates is the complement of the entries in the table, as there are only two classes. *Second*, the Initial-k and Random-k S-Vector approximations perform very well. This result is important because the Initial-k representation is the most practical one, as alluded to earlier.

It is also noteworthy that the performance of syntax-based schemes is relatively poor using the clustering outlier algorithm. There is one abnormal entry which is the clustering performance of S-V (all) in the "Chair vs. Staff" case, which most likely is due to overfitting.

6.4 Detecting Type 2b Anomalies and Data Harvesting Attacks

The focus here is on detecting syntactically similar queries, but differ in output data (data-values, output volume, or both). This is a significant query anomaly since, in a typical attack, a minor variation of a legitimate query can output a large volume of data to the attacker. This may go undetected and may be exploited for the purpose of data-harvesting. In other attack variations, the volume of the output may be typical, but the data values may be sensitive. These kinds of attacks fall into Type 2b in Table 2.

Test Setup: Since type-2b anomalous queries are not available from the real query set, we generate type-2b queries by slightly modifying the normal queries (i.e. queries normally executed by **GradVote** users). Thus, this generated "anomaly set" has approximately the same distribution as the normal queries. Anomalous queries are generated by varying arithmetic and logical operators and constants. As an example, consider the query

```
SELECT *
FROM vApplicants
WHERE reviewStatusID = 'a'
AND reviewStatusID = 'b';
```

can be slightly modified to become

```
SELECT *
FROM vApplicants
WHERE reviewStatusID = 'a'
OR reviewStatusID = 'b';
```

which yields a vastly different result set.

It must be noted that the queries considered here are different from masquerade attacks (since they are not representative of any authorized user of the system) and are thus not available for training *QStatProfiler*. Hence, supervised learning is not suitable here. We devise two detection techniques based on a single class of *normal* queries: *Cluster-Based Outlier Detection* based on Euclidean-distance clustering, and *Attrib-Deviation* which is a variation of clustering using the L_∞-norm as the distance function.

Cluster-based Outlier Detection: The set of queries encountered during the training phase are viewed as points in an m-dimensional Euclidean vector space, where m is the dimensionality of the S-vectors. For each user cluster, we select a point in the Euclidean space that is representative of the entire cluster, called the *cluster centroid*, which minimizes the sum of the squared Euclidean distances of the cluster points. For a test vector, the Euclidean distance from the cluster centroid is computed. The query is flagged as an outlier if the vector distance is greater than a specified threshold from any user. In our case, the threshold is chosen to be 3 times the standard deviation.

Attrib-Deviation: Consider, for example, that a user issues an anomalous query with a different statistic for the same attribute in the result schema as a normal query. In our representation, this difference shows up in one or more (depending on whether the attribute is *categoric* or *numeric*) dimensions of the S-Vector. Hence, monitoring for anomalies on *per-dimension* basis is a promising approach. Further, if a query generates unusual output for more than one attribute, this is likely to reflect in anomalous values for several S-Vector dimensions; thus, the number of anomalous dimensions for the S-Vector is a parameter that can be used for ranking potential query anomalies (i.e., queries with more anomalous S-Vector dimensions rank high as likely candidates for possible attacks). We utilize this approach for testing the custom-developed anomaly set – normal *Chair* and *Faculty* queries are used to compute the mean values of S-Vector attributes; three times the *standard-deviation* is again used as an anomaly separator.

A typical performance result with two user roles (*Chair* and *Faculty*) and corresponding anomalous query set is shown in Table 4.

With respect to the performance of the cluster-based outlier detection algorithm, a few points are worth noticing. As expected, the syntax-based schemes

show poor performance (since they are essentially 'blind' by design to the Type 2b anomalies). The detection rate for the S-Vector (all) is reasonable (83.87%). However, the Initial-k approximation's accuracy suffers significantly. Upon careful inspection, we find that many of the user queries make extensive use of the SQL *ORDER-BY* clause, which makes the Initial-k statistics unrepresentative of the overall result set statistics. This is ameliorated to some extent by the Random-k variation (e.g., for random $k = 20$, the detection rate improves to 67.7%); however, there is still a marked decline in performance indicating that the clustering scheme is sensitive to the approximation schemes and is affected negatively by them. Further analysis into the clustering reveals that this might not be a good choice for this type of anomaly detection. Although anomalies with significant variations in multiple dimensions are easily detected by clustering (as is the case with type-1 and type-2a anomalies), this may not be true with type-2b anomalies. Euclidean distances in high-dimensional space may be misleading indicators of anomalies because of the *curse of dimensionality*. For example, it is possible to have a highly anomalous value along a single dimension, which may not translate to a significant Euclidean cluster-distance (and *vice-versa*).

The results for Attrib-Deviation are much better. The syntax based schemes still perform poorly as expected. The data-centric schemes are much better, with detection rates close to 87%, better than the cluster-based schemes. The more important finding is that the attribute-deviation schemes are remarkably resilient to the approximation method. Both Initial-k and Random-k perform as well as the full vector representation; and the Initial-k performs unexpectedly well even with queries generating specific ordering of results.

The resiliency and accuracy of Attrib-Deviation can partially explained as follows. First, note that a single anomalous attribute in the result corresponds to variations in multiple dimensions of the S-Vector, each of which represents a statistical measurement. Also the extent of the anomaly may vary between result attributes (e.g., some attributes may have more atypical values). While a selective ordering (e.g., by SQL *ORDER-BY* clauses) may offer a skewed view of overall result statistics, the *Attrib-Deviation* technique operates on a per-attribute basis and is thus still able to identify anomalies. Secondly, many queries have more than one anomalous attribute; hence selective ordering may mask anomalies in some attributes, but not all of them. Thirdly, the selective ordering may not affect all statistical measurements of a single attribute equally (e.g., it may affect *Max*, but not *Median*). It is only when k is very low ($k = 5$) that initial-k performance drops, however Random-k as expected still offers reasonable performance.

We believe that the good performance of the Initial-k approximation with this detection technique has several practical implications. First, it indicates that a fast online anomaly detector can perform well by considering just a few (as long as it is not *too* few) initial output tuples. Randomized sampling of query results may not be feasible in general, especially for queries generating hundreds or thousands of output tuples (e.g., due to performance constraints), but our results here indicate that accuracy may not have to be sacrificed always

in the process of giving up random sampling. Further, we also believe that the S-Vector representation scheme and attribute-deviation based anomaly detection algorithm are quite resilient to attacks designed to mislead or bypass detection. It is very difficult for an attacker to craft queries so that multiple statistical measurements are controlled. A theoretical explanation of this intuition is an interesting research problem.

On the minus side, the false positive rates are still too high for the *Attrib-Deviation* schemes. Reducing the false-positive rates while maintaining/increasing the accuracy (true positive rates) is an important research question, which we plan to address in future work.

7 Concluding Remarks and Future Work

Queries: We construct the S-vectors by expressing the schema of each query result in terms of the attributes of the base schema. For select-project-join (SPJ) queries on base relations, the base schema is easily determined. When SPJ queries are also expressed on top of views, then we employed the view unfolding technique [33] to determine the base schema. View unfolding recursively replaces references to a view in a query expression with its corresponding view definition. For a class of queries larger than SPJ queries on base relations and views, it is not clear if the base schema can be determined. For example, union queries can map two different attributes in base relations into a single one in the query result, as the following example shows:

```
    SELECT g.name, g.gpa
    FROM GRADS g
UNION
    SELECT u.name, u.gpa
    FROM UGRADS u;
```

Here, there is no dimension in the S-vector to accommodate the first attribute of the query result. The same is true for computed attributes in results of complex (aggregation, group-by) queries. To accommodate such cases, we plan to investigate data provenance techniques [9] and revise the definition and the use of the S-vector accordingly.

Databases: The framework proposed in this paper assumes that the underlying database is *static*, i.e., there are no updates. Although this assumption is adequate for certain classes of databases (e.g., US census database), we plan to extend our work to *dynamic* databases. The first challenge is to determine if and when updates shift the boundary between normal and abnormal queries. If the database instance is updated significantly, then the classifiers become obsolete and two things need to be done: (a) detect when a phase shift occurs and re-train, and (b) adopt some form of re-enforcement and/or online learning.

For relatively less dynamic databases where updates are less frequent, such as OLAP databases that are heavily used for business intelligence and hence are good targets for insider attacks, it is possible to still apply the data-centric

approach, depending on the relative frequency between re-training and data updates. For instance, one can keep a history of legitimate user queries, re-execute them on the new data when the data changes are sufficiently heavy, and use the new result sets to re-train the machine learning model.

Another approach is to separate parts of the schema where data does not change very often and the part that does. Then, the data-centric approach can be applied to the "projection" of the data space which is static, and the syntax-centric approach can be applied to the dynamic part of the data. This separation can also be done automatically as one can keep track of the statistics of various attributes in the universal table. For example, attributes with high variation over time are more dynamic than others (e.g., Social Security numbers, bank accounts of existing customers, dates of births, addresses, and similar fields are mostly static attributes).

Activity context: In our approach, the context of a user's activity is a set of query results generated in the past by the same user or the group in which she belongs. We plan to investigate richer activity contexts and examine their effectiveness in detecting sophisticated attacks. Such contexts might include statistics of a user's session with the database, temporal and spatial correlations of the query results, and so on.

Performance: In cases where user queries return a significantly large number of results, computing statistics over the entire query result for anomaly detection is unacceptable from a performance standpoint. The initial-k approximation proposed in Section 6 can help improve performance without sacrificing too much accuracy. One potential drawback of this approach is that the queries in the training set might sort the results by a different attribute or in different order (ascending, descending) than an otherwise normal user query, thus leading to false positives. A possible solution to this problem is to choose one attribute of each base relation as the default order by attribute. Then, for every query in the training set add a *designated* ORDER BY clause that orders the result by the chosen attribute of the first base relation (alphabetically) used in the query. When a user query is submitted, the system submits a "shadow query" with the *designated* ORDER BY clause and uses this query result for detection.

Another source of performance improvement might be to design a new statistical model based on both the syntax-based and the data-centric approaches. In cases where we are relatively confident that the syntax-based approach gives a true positive, we may want to skip the data-centric engine altogether to avoid the database execution. In terms of accuracy, a good combined classifier might perform better too.

Although random-k does not markedly outperform initial-k in our experiments, we expect random-k to perform consistently for a wider range of datasets and queries. Of course, a problem that arises then is how to sample a query result without computing the complete result, given that RDBMSs follow the pipelined query execution model. For this hard problem, we plan to leverage

prior work on both SPJ queries [25,12] and queries for data analytics in the area of approximate query answering [16,3,5].

In conclusion, the techniques that we have presented and analyzed in this paper show significant potential as practical solutions for anomaly detection and insider threat mitigation in database systems.

References

1. Owasp top 10 2007 (2007), http://www.owasp.org/index.php/Top_10_2007
2. Owasp-sql injection prevention cheat sheet (2008), http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
3. Acharya, S., Gibbons, P.B., Poosala, V., Ramaswamy, S.: Join synopses for approximate query answering. In: SIGMOD Conference, pp. 275–286 (1999)
4. Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD 2000), pp. 439–450 (2000)
5. Babcock, B., Chaudhuri, S., Das, G.: Dynamic sample selection for approximate query processing. In: SIGMOD Conference, pp. 539–550 (2003)
6. Bishop, C.M.: Pattern Recognition and Machine Learning. Springer, Heidelberg (October 2007)
7. Bishop, M.: The insider problem revisited. In: Proc. of the 2005 Workshop on New Security Paradigms (NSPW 2005), pp. 75–76 (2005)
8. Brackney, R., Anderson, R.: Understanding the Insider Threat: Proceedings of a March 2004 Workshop. RAND Corp. (2004)
9. Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. In: ICDT, pp. 316–330 (2001)
10. Calvanese, D., Giacomo, G.D., Lenzerini, M.: On the decidability of query containment under constraints. In: Proc. of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 1998), pp. 149–158 (1998)
11. Cappelli, D.: Preventing insider sabotage: Lessons learned from actual attacks (2005), http://www.cert.org/archive/pdf/InsiderThreatCSI.pdf
12. Chaudhuri, S., Motwani, R., Narasayya, V.R.: On random sampling over joins. In: SIGMOD Conference, pp. 263–274 (1999)
13. Chung, C.Y., Gertz, M., Levitt, K.: Demids: a misuse detection system for database systems. In: Integrity and Internal Control Information Systems: Strategic Views on the Need for Control, pp. 159–178. Kluwer Academic Publishers, Norwell (2000)
14. CSO Magazine, US Secret Service, CERT, Microsoft: 2007 E-Crime Watch Survey (2007), http://www.sei.cmu.edu/about/press/releases/2007ecrime.html
15. Fonseca, J., Vieira, M., Madeira, H.: Online detection of malicious data access using dbms auditing. In: Proc. of the 2008 ACM Symposium on Applied Computing (SAC 2008), pp. 1013–1020 (2008)
16. Haas, P.J., Hellerstein, J.M.: Ripple joins for online aggregation. In: SIGMOD Conference, pp. 287–298 (1999)
17. Hu, Y., Panda, B.: Identification of malicious transactions in database systems. In: Proc. of the 7th International Database Engineering and Applications Symposium, pp. 329–335 (2003)
18. Kamra, A., Terzi, E., Bertino, E.: Detecting anomalous access patterns in relational databases. The VLDB Journal 17(5), 1063–1077 (2008)

19. Kenthapadi, K., Mishra, N., Nissim, K.: Simulatable auditing. In: Proc. of the ACM Symposium on Principles of Database Systems (PODS 2005), pp. 118–127 (2005)
20. Kruegel, C., Vigna, G.: Anomaly detection of web-based attacks. In: Proc. of the 10th ACM Conference on Computers and Communications Security (CCS 2003), pp. 251–261 (2003)
21. Lee, S.Y., Low, W.L., Wong, P.Y.: Learning fingerprints for a database intrusion detection system. In: Gollmann, D., Karjoth, G., Waidner, M. (eds.) ESORICS 2002. LNCS, vol. 2502, pp. 264–280. Springer, Heidelberg (2002)
22. Lee, V.C., Stankovic, J., Son, S.H.: Intrusion detection in real-time database systems via time signatures. In: Proc. of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000), p. 124 (2000)
23. Liu, P.: Architectures for intrusion tolerant database systems. In: Proc. of the 18th Annual Computer Security Applications Conference (ACSAC 2002), p. 311 (2002)
24. Maier, D., Ullman, J.D., Vardi, M.Y.: On the foundations of the universal relation model. ACM Trans. on Database Syst. 9(2), 283–308 (1984)
25. Olken, F., Rotem, D.: Simple random sampling from relational databases. In: VLDB, pp. 160–169 (1986)
26. Ramasubramanian, P., Kannan, A.: Intelligent multi-agent based database hybrid intrusion prevention system. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 393–408. Springer, Heidelberg (2004)
27. Roichman, A., Gudes, E.: Diweda – detecting intrusions in web databases. In: Proc. of the 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security, pp. 313–329 (2008)
28. Sandhu, R., Ferraiolo, D., Kuhn, R.: The nist model for role based access control. In: Proc. of the 5th ACM Workshop on Role Based Access Control (2000)
29. Schneier, B.: Secrets and Lies: Digital Security in a Networked World. John Wiley and Sons, New York (2000)
30. Schonlau, M., DuMouchel, W., Ju, W., Karr, A., Theus, M., Vardi, Y.: Computer intrusion: Detecting masquerades. Statistical Science 16(1), 58–74 (2001)
31. Spalka, A., Lehnhardt, J.: A comprehensive approach to anomaly detection in relational databases. In: DBSec, pp. 207–221 (2005)
32. Srivastava, A., Sural, S., Majumdar, A.K.: Database intrusion detection using weighted sequence mining. Journal of Computers 1(4), 8–17 (2006)
33. Stonebraker, M.: Implementation of integrity constraints and views by query modification. In: SIGMOD Conference, pp. 65–78 (1975)
34. Valeur, F., Mutz, D., Vigna, G.: A learning-based approach to the detection of sql attacks. In: Julisch, K., Krügel, C. (eds.) DIMVA 2005. LNCS, vol. 3548, pp. 123–140. Springer, Heidelberg (2005)
35. Wenhui, S., Tan, D.: A novel intrusion detection system model for securing web-based database systems. In: Proc. of the 25th International Computer Software and Applications Conference on Invigorating Software Development (COMPSAC 2001), p. 249 (2001)
36. Yao, Q., An, A., Huang, X.: Finding and analyzing database user sessions. In: Proc. of Database Systems for Advanced Applications, pp. 283–308 (2005)

Privilege States Based Access Control for Fine-Grained Intrusion Response

Ashish Kamra[1] and Elisa Bertino[2]

[1] School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA
akamra@purdue.edu
[2] School of Computer Science, Purdue University, West Lafayette, IN, USA
bertino@cs.purdue.edu

Abstract. We propose an access control model specifically developed to support fine-grained response actions, such as request suspension and request tainting, in the context of an anomaly detection system for databases. To achieve such response semantics, the model introduces the concept of *privilege states* and *orientation modes* in the context of a role-based access control system. The central idea in our model is that privileges, assigned to a user or role, have a state attached to them, thereby resulting in a *privilege states based access control* (PSAC) system. In this paper, we present the design details and a formal model of PSAC tailored to database management systems (DBMSs). PSAC has been designed to also take into account role hierarchies that are often present in the access control models of current DBMSs. We have implemented PSAC in the PostgreSQL DBMS and in the paper, we discuss relevant implementation issues. We also report experimental results concerning the overhead of the access control enforcement in PSAC. Such results confirm that our design and algorithms are very efficient.

1 Motivation

An access control mechanism is typically based on the notion of authorizations. An authorization is traditionally characterized by a three-element tuple of the form $< A, R, P >$ where A is the set of permissible actions, R is the set of protected resources, and P is the set of principals. When a principal tries to access a protected resource, the access control mechanism checks the rights (or privileges) of the principal against the set of authorizations in order to decide whether to allow or deny the access request.

The main goal of this work is to extend the decision semantics of an access control system beyond the *all-or-nothing* allow or deny decisions. Specifically, we provide support for more *fine-grained* decisions of the following two forms: *suspend*, wherein further negotiation (such as a second factor of authentication) occurs with the principal before deciding to allow or deny the request, and *taint*, that allows one to audit the request in-progress, thus resulting in further monitoring of the principal, and possibly in the suspension or dropping of subsequent requests by the same principal. The main motivation for proposing such fine-grained access check decisions is to provide system

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 402–421, 2010.

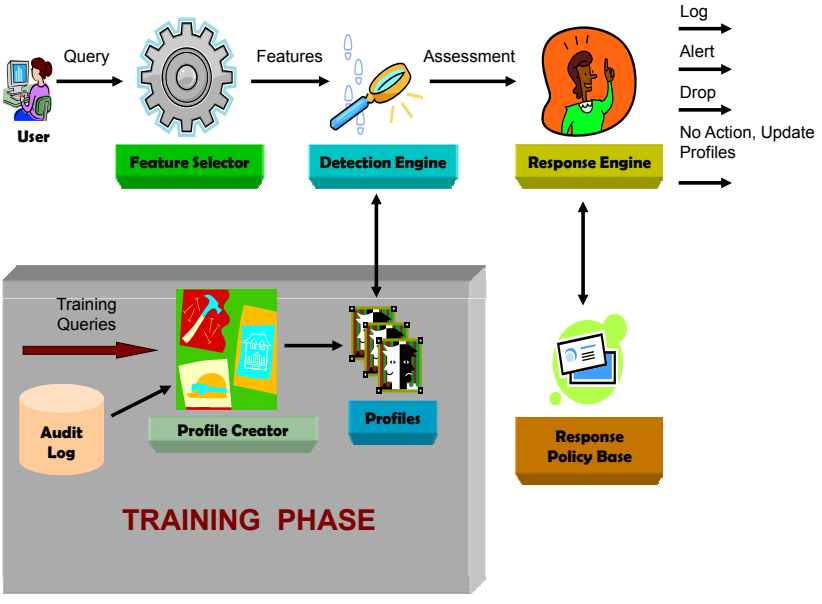


Fig. 1. Anomaly Detection and Response System Architecture

support for extending the response action semantics of an application level anomaly detection (AD) system that detects the anomalous patterns of requests submitted to it.

Consider the architecture of a database specific AD mechanism using fine-grained response actions as shown in Figure 1 [8,15,14]. The system consists of three main components: the traditional database server that handles the query execution, the profile creator module for creating user profiles from the training data, and the anomaly detection and response mechanisms integrated with the core database functionality. The flow of interactions for the anomaly detection and response process is as follows: During the training phase, the SQL commands submitted to the database (or read from the audit log) are analyzed by the profile creator module to create the initial profiles of the database users. In the detection phase, for every SQL command under detection, the feature selector module extracts the features from the queries in the format expected by the detection engine. The detection engine then runs the extracted features through the detection algorithm. If an anomaly detected, the detection mechanism submits its assessment of the SQL command to the response engine according to a pre-defined interface; otherwise the command information is sent to the profile creator process for updating the profiles.

The response engine consults a base of response policies to issue a suitable response action depending on the assessment of the anomalous query submitted by the detection engine. The system supports three types of response actions, that we refer to respectively as conservative actions, fine-grained actions, and aggressive actions. The conservative actions, such as sending an alert, allow the anomalous request to go through, whereas the aggressive actions can effectively block the anomalous request.

Fine-grained response actions, supported by the extended decision semantics of our access control mechanism, are neither conservative nor aggressive. Such actions may result in either request suspension (supported by the *suspend* decision semantics) and request tainting (supported by the *taint* decision semantics).

Why do we need to extend the access control mechanism to support such response actions? Certainly, such responses may also be issued by an AD mechanism working independently of the underlying access control system. The usefulness of our approach is evident from the following scenario. Suppose we model a user request as the usage of a set of *privileges* in the system where a privilege is defined as an operation on a resource. For example, the SQL query '*SELECT * FROM orders, parts*' is modeled as using the privileges {select,orders} and {select,parts} in the context of a database management system (DBMS). After detecting such request as anomalous (using any anomaly detection algorithm), consider that we want to re-authenticate the user and drop the request in case the re-authentication procedure fails. Suppose that every time a similar request is detected to be anomalous, we want the same re-authentication procedure to be repeated. If our response mechanism does not *remember* the requests, then the request will always undergo the detection procedure, detected to be anomalous and then submitted to the response mechanism to trigger the re-authentication procedure. A more generic and flexible approach for achieving such response semantics is to *attach a suspend state to the privileges* associated with the anomalous request. Then for every subsequent similar request (that uses the same set of privileges as the earlier request that was detected to be anomalous), the semantics of the privilege in the *suspend* state automatically triggers the re-authentication sequence of actions for the request under consideration without the request being subjected to the detection mechanism. Moreover, if the system is set-up such that the request is always subjected to the detection mechanism (in case access control enforcement is performed after the intrusion detection task), more advanced response logic can be built based on the fact that a request is detected to be anomalous whose privileges are already in the *suspend* state.

In addition to supporting fine-grained intrusion response, manually moving a privilege to the *suspend* state (using administrative commands) provides the basis for an event based continuous authentication mechanism. Similar arguments can be made for attaching the *taint* state to a privilege that triggers auditing of the request in progress. Since we extend the decision semantics of our access control system using privilege states, we call it a *privilege state based access control* (PSAC) system. For the completeness of the access control decisions, a privilege, assigned to a user or role, in PSAC can exist in the following five states: *unassign*, *grant*, *taint*, *suspend*, and *deny*. The privilege states, the state transition semantics and a formal model of PSAC are described in detail in Section 2. Note that the PSAC model that we present in Section 2 is flexible enough to allow more than the above mentioned five states.

We have developed PSAC in the context of a role based access control (RBAC) system [18]. Extending PSAC with roles presents the main challenge of *state conflict resolution*, that is, deciding on the final state of a privilege when a principal receives the same privilege in different states from other principals. Moreover, additional complexity is introduced when the roles are arranged in a hierarchy where the roles higher-up in the

hierarchy inherit the privileges of the lower level roles. We present precise semantics in PSAC to deal with such scenarios.

The main contributions of this paper can be summarized as follows:

1. We present the design details, and a formal model of PSAC in the context of a DBMS.
2. We extend the PSAC semantics to take into account a role hierarchy.
3. We implement PSAC in the PostgreSQL DBMS [5] and discuss relevant design issues.
4. We conduct an experimental evaluation of the access control enforcement overhead introduced by the maintenance of privilege states in PSAC, and show that our implementation design is very efficient.

The rest of the paper is organized as follows. Section 2 presents the details of PSAC and its formal model; it also discusses how a role hierarchy is supported. Section 3 presents the details of the system implemented in PostgreSQL, and the experimental results concerning the overhead introduced by the privilege states on the access control functions. Section 4 discusses the related work in this area. We conclude the paper in Section 5.

2 PSAC Design and Formal Model

In this section, we introduce the design and the formal model underlying PSAC. We assume that the authorization model also supports roles, in that RBAC is widely used by access control systems of current DBMSs [11,4,7]. In what follows, we first introduce the privilege state semantics and state transitions. We then discuss in detail how those notions have to be extended when dealing with role hierarchies.

2.1 Privilege States Dominance Relationship

PSAC supports five different privilege states that are listed in Table 1. For each state, the table describes the semantics in terms of the result of an access check.

A privilege in the *unassign* state is equivalent to the privilege not being assigned to a principal; and a privilege in the *grant* state is equivalent to the privilege being

Table 1. Privilege States

State	Access Check Result Semantics
unassign	The access to the resource is not granted.
grant	The access to the resource is granted.
taint	The access to the resource is granted; the system audits access to the resource.
suspend	The access to the resource is not granted until further negotiation with the principal is satisfied.
deny	The access to the resource is not granted.

granted to a principal. We include the *deny* state in our model to support the concept of negative authorizations in which a privilege is specifically denied to a principal [9]. The *suspend* and the *taint* states support the fine-grained decision semantics for the result of an access check.

In most DBMSs, there are two distinct ways according to which a user/role[1] can obtain a privilege p on a database object o:

1. **Role-assignment:** the user/role is assigned a role that has been assigned p;
2. **Discretionary:** the user is the owner of o; or the user/role is assigned p by another user/role that has been assigned p with the GRANT option[2].

Because of the multiple ways by which a privilege can be obtained, conflicts are natural in cases where the same privilege, obtained from multiple sources, exists in different states. Therefore, a *conflict resolution* strategy must be defined to address such cases. Our strategy is to introduce a *privilege states dominance* (PSD) relation (see Figure 2). The PSD relation imposes a total order on the set of privilege states such that any two states are comparable under the PSD relation. Note the following characteristics of the semantics of the PSD relation. First, the *deny* state overrides all the other states to support the concept of a negative authorization [9]. Second, the *suspend*, and the *taint* states override the *grant* state as they can be triggered as potential response actions to an anomalous request. Finally, the *unassign* state is overridden by all the other states thereby preserving the traditional semantics of privilege assignment.

The PSD relation is the core mechanism that PSAC provides for resolving conflicts. For example, consider a user u that derives its privileges by being assigned a role r. Suppose that a privilege p is assigned to r in the *grant* state. Now suppose we directly deny p to u. The question is which is the state of privilege p for u, in that u has received p with two different states. We resolve such conflicts in PSAC using the PSD relation. Because in the PSD relation, the *deny* state overrides the *grant* state, p is denied to u.

We formally define a PSD relation as follows:

Definition 1. (**PSD Relation**) Let n be the number of privilege states. Let $S = \{s_1, s_2 \ldots s_n\}$ be the set of privilege states. The PSD relation is a binary relation (denoted by $\preceq$) on S such that for all $s_i, s_j, s_k \in S$:

1. $s_i \preceq s_j$ means s_i overrides s_j
2. if $s_i \preceq s_j$ and $s_j \preceq s_i$, then $s_i = s_j$ (anti-symmetry)
3. if $s_i \preceq s_j$ and $s_j \preceq s_k$, then $s_i \preceq s_k$ (transitivity)
4. $s_i \preceq s_j$ or $s_j \preceq s_i$ (totality) □

2.2 Privilege State Transitions

We now turn our attention to the privilege state transitions in PSAC. Initially, when a privilege is not assigned to a principal, it is in the *unassign* state for that principal. Thus,

[1] From here on, we use the terms *principal* and *user/role* interchangeably.

[2] A privilege granted to a principal with the GRANT option allows the principal to grant that privilege to other principals [2].

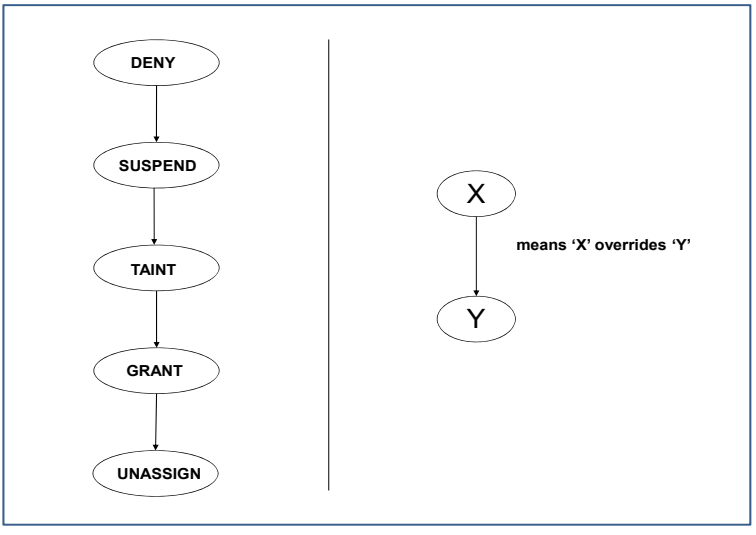


Fig. 2. Privilege States Dominance Relationship

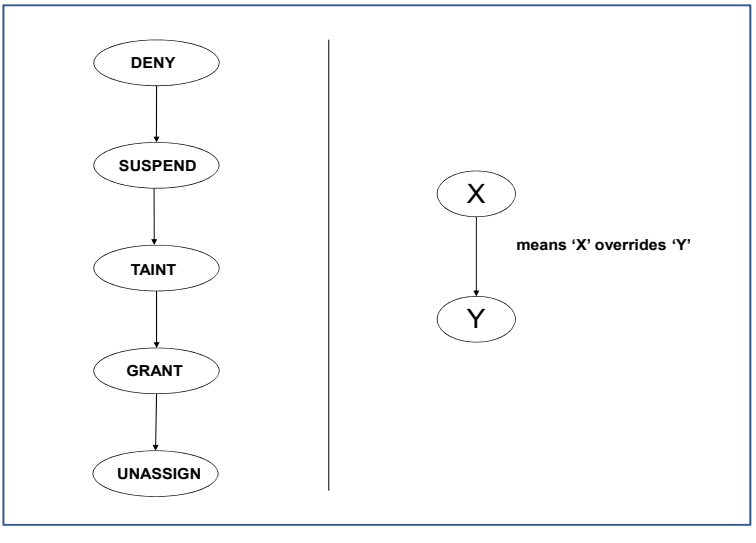


Fig. 3. Privilege State Transitions

the *unassign* state is the default (or initial) state of a privilege. The state transitions can be triggered as internal response actions by an AD system, or as ad-hoc administrative commands. In what follows, we discuss the various administrative commands available in PSAC to trigger privilege state transitions.

The GRANT command is used to assign a privilege to a principal in the *grant* state whereas the REVOKE command is used to assign a privilege to a principal in the *unassign* state. In this sense, these commands support similar functionality as the SQL-99 GRANT and REVOKE commands [2]. The DENY command assigns a privilege to a principal in the *deny* state. We introduce two new commands in PSAC namely, SUSPEND and TAINT, for assigning a privilege to a principal in the *suspend* and the *taint* states, respectively. The privilege state transitions are summarized in Figure 3. Note the constraint that a privilege assigned to a principal on a DBMS object can only exist in one state at any given point in time.

2.3 Formal Model

In this section, we formally define the privilege model for PSAC in the context of a DBMS. The model is based on the following relations and functions:

Relations

1. U, the set of all users in the DBMS.
2. R, the set of all roles in the DBMS.
3. $PR = U \cup R$, the set of principals (users/roles) in the DBMS.
4. OT, the set of all DBMS object types such as *server*, *database*, *schema*, *table*, and so forth.
5. O, the set of all DBMS objects of all object types.
6. OP, the set of all operations defined on the object types in OT, such as *select*, *insert*, *delete*, *drop*, *backup*, *disconnect*, and so forth.
7. $S = \{$*deny,suspend,taint,grant,unassign*$\}$, a totally ordered set of privilege states under the PSD relation (Definition 2.1).
8. $P \subset OP \times OT$, a many-to-many relation on operations and object types representing the set of all privileges. Note that not all operations are defined for all object types. For example, tuples of the form $(select, server)$ or $(drop, server)$ are not elements of P.
9. $URA \subseteq U \times R$, a many-to-many user to role assignment relation.
10. $PRUPOSA \subset PR \times U \times P \times O \times S$, a principal to user to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to users (assignees) by other principals (assigners) on DBMS objects[3].
11. $PRRPOSA \subset PR \times R \times P \times O \times S$, a principals to role to privilege to object to state assignment relation. This relation captures the state of the access control mechanism in terms of the privileges, and their states, that are directly assigned to roles (assignees) by principals (assigners).

[3] In PSAC, a role can also be an assigner of privileges. Consider a situation when a user u gets a privilege p (with grant option) through assignment of role r. If u grants p to some other user u', PSAC records p as being granted to u' by r even though the actual GRANT command was executed by u.

These relations capture the state of the access control system in terms of the privilege and the role assignments. The functions defined below determine the state of a privilege assigned to a user/role on a DBMS object.

Functions

1. $assigned\_roles(u) : U \rightarrow 2^R$, a function mapping a user u to its assigned roles such that $assigned\_roles(u) = \{r \in R \mid (u, r) \in URA\}$. This function returns the set of roles that are assigned to a user.
2. $priv\_states(pr, r', p, o) : PR \times R \times P \times O \rightarrow 2^S$, a function mapping a principal pr (privilege assigner), a role r', a privilege p, and an object o to a set of privilege states such that $priv\_states(pr, r', p, o) = \{s \in S \mid (pr, r', p, o, s) \in PRRPOSA\}$. This function returns the set of states for a privilege p, that is directly assigned to the role r' by the principal pr, on an object o.
3. $priv\_states(pr, u', p, o) : PR \times U \times P \times O \rightarrow 2^S$, a function mapping a principal pr (privilege assigner), a user u', a privilege p, and an object o to a set of privilege states such that $priv\_states(pr, u', p, o) = \{s \in S \mid (pr, u', p, o, s) \in PRUPOSA\} \cup_{r \in assigned\_roles(u')} priv\_states(pr, r, p, o)$. The set of states returned by this function is the union of the privilege state directly assigned to the user u' by the principal pr, and the privilege states (also assigned by pr) obtained through the roles assigned to u'.
4. $priv\_states(r, p, o) : R \times P \times O \rightarrow 2^S$, a function mapping a role r, a privilege p, and an object o to a set of privilege states such that $priv\_states(r, p, o) = \cup_{pr \in PR} priv\_states(pr, r, p, o)$. This function returns the set of states for a privilege p, that is directly assigned to the role r by any principal in the DBMS, on an object o.
5. $priv\_states(u', p, o) : U \times P \times O \rightarrow 2^S$, a function mapping a user u', a privilege p, and an object o to a set of privilege states such that $priv\_states(u', p, o) = \cup_{pr \in PR} priv\_states(pr, u', p, o)$. This function returns the set of states for a privilege p, that is directly assigned to the user u' by any principal in the DBMS, on an object o.
6. $PSD\_state(2^S) : 2^S \rightarrow S$, a function mapping a set of states 2^S to a state $\in S$ such that $PSD\_state(2^S) = s' \in 2^S \mid \forall_{s \in 2^S \mid s \neq s'} \; s' \preceq s$. This function returns the final state of a privilege using the PSD relation.

2.4 Role Hierarchy

Traditionally, roles can be arranged in a conceptual hierarchy using the role-to-role assignment relation. For example, if a role r_2 is assigned to a role r_1, then r_1 becomes a parent of r_2 in the conceptual role hierarchy. Such hierarchy signifies that the role r_1 inherits the privileges of the role r_2 and thus, is a more *privileged* role then r_2. However, in PSAC such privilege inheritance semantics may create a problem because of a $deny/suspend/taint$ state attached to a privilege. The problem is as follows. Suppose a privilege p is assigned to the role r_2 in the *deny* state. The role r_1 will also have such privilege in the *deny* state since it inherits it from the role r_2. Thus, denying

a privilege to a lower level role has the affect of denying that privilege to all roles that inherit from that role. This defeats the purpose of maintaining a role hierarchy in which roles higher up the hierarchy are supposed to be more privileged than the descendant roles. To address this issue, we introduce the concept of *privilege orientation*. We define three privilege orientation modes namely, *up, down*, and *neutral*. A privilege assigned to a role in the *up* orientation mode means that the privilege is also assigned to its parent roles. On the other hand, a privilege assigned to a role in the *down* orientation mode means that the privilege is also assigned to its children roles; while the *neutral* orientation mode implies that the privilege is neither assigned to the parent roles nor to the children roles. We put the following two constraints on the assignment of orientation modes on the privileges.

- A privilege assigned to a role in the *grant* or in the *unassign* state is always in the *up* orientation mode thereby maintaining the traditional privilege inheritance semantics in a role hierarchy.
- A privilege assigned to a role in the *deny, taint*, or *suspend* state may only be in the *down* or in the *neutral* orientation mode. Assigning such privilege states to a role in the *down* or *neutral* mode ensures that the role still remains more privileged than its children roles. In addition, the *neutral* mode is particularly useful when a privilege needs to be assigned to a role without affecting the rest of the role hierarchy (when responding to an anomaly, for example).

We formalize the privilege model of PSAC in the presence of a role hierarchy as follows:

1. $RRA \subset R \times R$, a many-to-many role to role assignment relation. A tuple of the form $(r_1, r_2) \in R \times R$ means that the role r_2 is assigned to the role r_1. Thus, role r_1 is a parent of role r_2 in the conceptual role hierarchy.
2. $OR = \{up, down, neutral\}$, the set of privilege orientation modes.
3. $PRRPOSORA \subset PR \times R \times P \times O \times S \times OR$, a principal to role to privilege to object to state to orientation mode assignment relation. This relation captures the state of the access control system in terms of the privileges, their states, and their orientation modes that are directly assigned to roles by principals.
4. $assigned\_roles(r') : R \rightarrow 2^R$, a function mapping a role r' to its assigned roles such that $assigned\_roles(r') = \{r \in R \mid (r', r) \in RRA\} \cup assigned\_roles(r)$. This function returns the set of the roles that are directly and indirectly (through the role hierarchy) assigned to a role; in other words, the set of descendant roles of a role in the role hierarchy.
5. $assigned\_roles(u) : U \rightarrow 2^R$, a function mapping a user u to its assigned roles such that $assigned\_roles(u) = \{r \in R \mid (u, r) \in URA\} \cup assigned\_roles(r)$. This function returns the set of roles that are directly and indirectly (through the role hierarchy) assigned to a user.
6. $assigned\_to\_roles(r') : R \rightarrow 2^R$, a function mapping a role r' to a set of roles such that $assigned\_to\_roles(r') = \{r \in R \mid (r, r') \in RRA\} \cup assigned\_to\_roles(r)$. This function returns the set of roles that a role is directly and indirectly (through

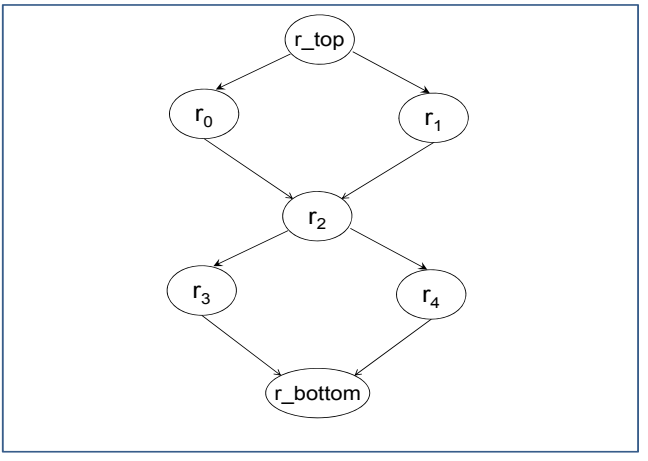


Fig. 4. A Sample Role Hierarchy

the role hierarchy) assigned to; in other words, the set of ancestor roles of a role in the role hierarchy.

We redefine the $priv\_states(pr, r', p, o)$ function in the presence of a role hierarchy taking into account the privilege orientation constraints as follows:

7. $priv\_states(pr, r', p, o) : PR \times R \times P \times O \rightarrow 2^S$, a function mapping a principal pr, a role r', a privilege s, and an object o to a set of privilege states such that $priv\_states(pr, r', p, o) = \{s \in S \mid \forall\ or \in OR,\ (pr, r', p, o, s, or) \in PRRPOSORA\ \} \cup \{s \in \{grant, unassign\} \mid \forall\ r \in assigned\_roles(r'),\ (pr, r, p, o, s, \text{`}up') \in PRRPOSORA\ \} \cup \{s \in \{deny, suspend, taint\} \mid \forall\ r \in assigned\_to\_roles(r'),\ (pr, r, p, o, s, \text{`}down') \in PRRPOSORA\ \}$. The set of privilege states returned by this function is the union of the privilege states directly assigned to the role r' by the principal pr, the privilege states in the *grant* or the *unassign* states (also assigned by pr) obtained through the descendant roles of r', and the privilege states in the *deny*, *suspend*, and *taint* states (also assigned by pr) obtained through the roles that are the ancestor roles of r', and that are in the *down* orientation mode.

We now present a comprehensive example of the above introduced relations and functions in PSAC. Consider a sample role hierarchy in Figure 4. Table 2 shows the state of a sample $PRRPOSORA$ relation.

Table 2. $PRRPOSORA$ relation

PR	R	P	O	S	OR
SU_1	r_top	$select$	t_1	$deny$	$neutral$
SU_1	r_0	$select$	t_1	$taint$	$down$
SU_1	r_bottom	$select$	t_1	$grant$	up
SU_2	r_top	$select$	t_1	$suspend$	$down$

Let the role r_2 be assigned to the user u_1. To determine the final state of the $select$ privilege on the table t_1 for the user u_1, we evaluate $priv\_states(u_1, select, t_1)$ as follows:

$$
\begin{aligned}
&priv\_states(u_1, select, t_1)\\
&= priv\_states(SU_1, u_1, select, t_1) \cup\\
&\quad priv\_states(SU_2, u_1, select, t_1)\\
&= priv\_states(SU_1, r_2, select, t_1) \cup\\
&\quad priv\_states(SU_2, r_2, select, t_1)\\
&= \{taint\} \cup\\
&\quad \{grant\} \cup \{suspend\}\\
&= \{taint, grant, suspend\}
\end{aligned}
$$

The final state is determined using the $PSD\_state()$ function as follows:

$$PSD\_state(taint, grant, suspend) = suspend$$

3 Implementation and Experiments

In this section, we present the details on how to extend a real-world DBMS with PSAC. We choose to implement PSAC in the PostgreSQL 8.3 open-source object-relational DBMS [5]. In the rest of the section, we use the term PSAC:PostgreSQL to indicate PostgreSQL extended with PSAC, and BASE:PostgreSQL to indicate the official PostgreSQL 8.3 release. The implementation of PSAC:PostgreSQL has to meet two design requirements. The first requirement is to maintain backward compatibility of PSAC:PostgreSQL with BASE:PostgreSQL. We intend to release PSAC:PostgreSQL for general public use in the near future; therefore it is important to take into account the backward compatibility issues in our design. The second requirement is to minimize the overhead for maintaining privilege states in the access control mechanism. In particular, we show that the time taken for the access control enforcement code in the presence of privilege states is not much higher than the time required by the access control mechanism of BASE:PostgreSQL. In what follows, we first present the design details of PSAC:PostgreSQL, and then we report experimental results showing the efficiency of our design.

3.1 PSAC:PostgreSQL

Access control in BASE:PostgreSQL is enforced using access control lists (ACLs). Every DBMS object has an ACL associated with it. An ACL in BASE:PostgreSQL is a one-dimensional array; the elements of such an array have values of the *ACLItem* data type. An ACLItem is the basic unit for managing privileges of an object. An ACLItem is implemented as a structure with the following fields: *granter*, the user/role granting the privileges; *grantee*, the user/role to which the privileges are granted; and *privs*, a 32 bit integer (on 32 bit machines) managed as a bit-vector to indicate the privileges granted

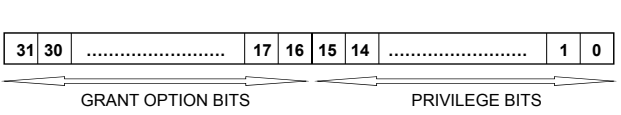


Fig. 5. ACLItem privs field

to the grantee. A new ACLItem is created for every unique pair of granter and grantee. There are 11 pre-defined privileges in BASE:PostgreSQL with a bit-mask associated with each privilege [6]. As shown in Figure 5, the lower 16 bits of the privs field are used to represent the granted privileges, while the upper 16 are used to indicate the $grant$ option[4]. If the k^{th} bit is set to 1 ($0 \leq k < 15$), privilege p_k is granted to the user/role. If the $(k + 16)^{th}$ bit is also set to 1, then the user/role has the grant option on privilege p_k.

Design Details. There are two design options to extend BASE:PostgreSQL to support privilege states. The first option is to extend the ACLItem structure to accommodate privilege states. The second option is to maintain the privilege states in a separate data structure. We chose the latter option. The main reason is that we want to maintain backward compatibility with BASE:PostgreSQL. Extending the existing data structures can introduce potential bugs at other places in the code base that we want to avoid. In BASE:PostgreSQL, the pg_class system catalog is used to store the metadata information for database objects such as tables, views, indexes and sequences. This catalog also stores the ACL for an object in the *acl* column that is an array of ACLItems. We extend the pg_class system catalog to maintain privilege states by adding four new columns namely: the *acltaint* column to maintain the tainted privileges; the *aclsuspend* column to maintain the suspended privileges; the *acldeny* column to maintain the denied privileges; and the *aclneut* column to indicate if the privilege is in the *neutral* orientation mode. Those state columns and the *aclneut* column are of the same data type as the *acl* column, that is, an array of ACLItems. The lower 16 bits of the privs field in those state and *aclneut* columns are used to indicate the privilege states and the orientation mode respectively. This strategy allows us to use the existing privilege bit-masks for retrieving the privilege state and orientation mode from these columns. The upper 16 bits are kept unused. Table 3 is the truth table capturing the semantics of the privs field bit-vector in PSAC:PostgreSQL.

Authorization Commands. We have modified the BASE:PostgreSQL GRANT and REVOKE authorization commands to implement the privilege state transitions. In addition, we have defined and implemented in PSAC:PostgreSQL three new authorization commands, that is, the DENY, the SUSPEND, and the TAINT commands. As discussed in the Section 2, the DENY command moves a privilege to the *deny* state, the SUSPEND command moves a privilege to the *suspend* state, and the TAINT command moves a privilege to the *taint* state. The default privilege orientation mode for these

[4] Recall that the grant option is used to indicate that the granted privilege may be granted by the grantee to other users/roles.

Table 3. Privilege States/Orientation Mode Truth Table for the privs Field in PSAC:PostgreSQL

acl k^{th} bit	**acl taint** k^{th} bit	**acl suspend** k^{th} bit	**acl deny** k^{th} bit	**acl neut** k^{th} bit	p_k **state**
0	0	0	0	0	unassign/up
1	0	0	0	0	grant/up
0	1	0	0	0	taint/down
0	0	1	0	0	suspend/down
0	0	0	1	0	deny/down
0	1	0	0	1	taint/neutral
0	0	1	0	1	suspend/neutral
0	0	0	1	1	deny/neutral
Rest all other combinations are not allowed by the system.					

Table 4. New Authorization Commands in PSAC:PostgreSQL

TAINT {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]
SUSPEND {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]
DENY {privilege name(s) | ALL} ON {object name(s)} TO {user/role name(s) | PUBLIC} [NEUT ORNT]

commands is the *down* mode with the option to override that by specifying the *neutral* orientation mode. The administrative model for these commands is similar to that of the SQL-99 GRANT command, that is, a DENY/SUSPEND/TAINT command can be executed on privilege p for object o by a user u iff u has the grant option set on p for o or u is the owner of o. The syntax for the commands is reported in Table 4. Note that in the current version of PSAC:PostgreSQL, the new commands are applicable on the database objects whose metadata are stored in the pg_class system catalog.

Access Control Enforcement. We have instrumented the access control enforcement code in BASE:PostgreSQL with the logic for maintaining the privilege states and orientation modes. The core access control function in BASE:PostgreSQL returns a true/false output depending on whether the privilege under check is granted to the user or not. In contrast, the core access control function in PSAC:PostgreSQL returns the final state of the privilege to the calling function. The calling function then executes a pre-configured action depending upon the state of the privilege. As a proof of concept, we have implemented a re-authentication procedure in PSAC:PostgreSQL when a privilege is in the *suspend* state. The re-authentication procedure is as follows:

Re-authentication Procedure. Recall that when a privilege is in the $suspend$ state, further negotiation with the end-user must be satisfied before the user-request is executed by the DBMS. In the current version of PSAC, we implement a procedure that re-authenticates the user if a privilege, after applying the PSD relationship, is found in

the *suspend* state. The re-authentication scheme is as follows. In BASE:PostgreSQL, an authentication protocol is carried out with the user whenever a new session is established between a client program and the PostgreSQL server. In PSAC:Postgresql, the same authentication protocol is replayed in the middle of a transaction execution when access control enforcement is in progress, and a privilege is found in the *suspend* state. We have modified the client library functions of BASE:PostgreSQL to implement such protocol in the middle of a transaction execution. If the re-authentication protocol fails, the user request is dropped. If it succeeds, the request proceeds as usual, and no changes are made to the state of the privilege. Note that such re-authentication procedure scheme is implemented as a proof-of-concept in PSAC:Postgresql. More advanced forms of actions such as a second-factor of authentication can also be implemented.

Access Control Enforcement Algorithm. The pseudo-code for the access control enforcement algorithm in PSAC:PostgreSQL is presented in the Listing 1. The function *aclcheck()* takes as input a privilege in_priv - whose state needs to be determined, a database object in_object - that is the target of a request, and a user in_user - the user exercising the usage of in_priv. The output of the algorithm is the state of the in_priv. The algorithm proceeds as follows. Since we define a total order on the privilege states, it is sufficient to check each state in the order of its rank in the PSD relation (cfr. Section 2). Thus, we first check for the existence of in_priv in the *deny* state, followed by the *suspend* state, the *taint* state, and then the *grant* state. The function for checking the state of in_priv (function *check_priv()*) in an Acl is designed to take into account all the roles that are directly and indirectly (through a role hierarchy) assigned to the in_user. Note that most expensive operation in the *check_priv()* function is the run-time inheritance check of roles, that is, to check whether the *user_role* is an ancestor or descendant of the *acl_role* (lines 58 and 62). We make such check a constant time operation in our implementation by maintaining a cache of the assigned roles for every user/role in the DBMS. Thus, the running time of the access control enforcement algorithm is primarily dependent upon the sizes of various Acls.

If the privilege is not found to be in the above mentioned states, the *unassign* state is returned as the output of the access check algorithm.

```
1 ------------------------------------------------------------
2 Input
3 in_user    : The user executing the command
4 in_object  : Target database object
5 in_priv    : Privilege to check
6
7 Output
8 The privilege state
9 ------------------------------------------------------------
10 function aclcheck(in_user, in_object, in_priv) returns state
11 {
12   //Get the neutral orientation ACL for in_object
13   NeutACL = get_neut_ornt(in_object);
```

```
14
15   //Deny if in_user has in_priv in DENY state
16   DenyACL = get_deny_state_acl(in_object);
17   if (check_priv(in_priv,DenyACL,in_user,NeutACL,DENY) == true)
18       return DENY;
19
20   //Suspend if in_user has in_priv in SUSPEND state
21   SuspendACL = get_suspend_state_acl(in_object);
22   if (check_priv(in_priv,SuspendACL,in_user,NeutACL,SUSPEND) ==
         true)
23       return SUSPEND;
24
25   //Taint if in_user has in_priv in TAINT state
26   TaintACL = get_taint_state_acl(in_object);
27   if (check_priv(in_priv,TaintACL,in_user,NeutACL,TAINT) == true
         )
28      return TAINT;
29
30   //Grant if in_user has in_priv in GRANT state
31   GrantACL = get_grant_state_acl(in_object);
32   if (check_priv(in_priv,GrantACL,in_user,NeutACL,GRANT) == true
         )
33      return GRANT;
34
35   //Else return UNASSIGN state
36   return UNASSIGN;
37 }
38 ------------------------------------------------------------
39 function check_priv(in_priv,AclToCheck,in_user,NeutACL,
       state_to_check)
40 returns boolean
41 {
42   //First, perform the inexpensive step of checking the
         privileges directly assigned to the in_user
43   if (in_user has in_priv in AclToCheck)
44     return true;
45
46   //Get all the roles directly assigned to in_user
47   user_role_list = get_roles(in_user);
48
49   //Do the following for every role directly assigned to in_user
50   for each user_role in user_role_list
51   {
52     //Do the following for every role entry in AclToCheck
53     for each acl_role in AclToCheck
54     {
55       if (state_to_check == GRANT)
56       {
```

```
57          //Orientation of privileges in GRANT state is UP
58          if ((user_role == acl_role OR user_role is an ANCESTOR
                of acl_role) AND acl_role has in_priv)
59
60            return true;
61        }
62        else if ((user_role == acl_role OR user_role is a
              DESCENDANT of acl_role) AND acl_role has in_priv)
63        {
64          if (acl_role has in_priv in NeutACL)
65            continue looping through AclToCheck;
66          else
67            return true;
68        }
69      }
70    }
71
72    return false;
73  }
```

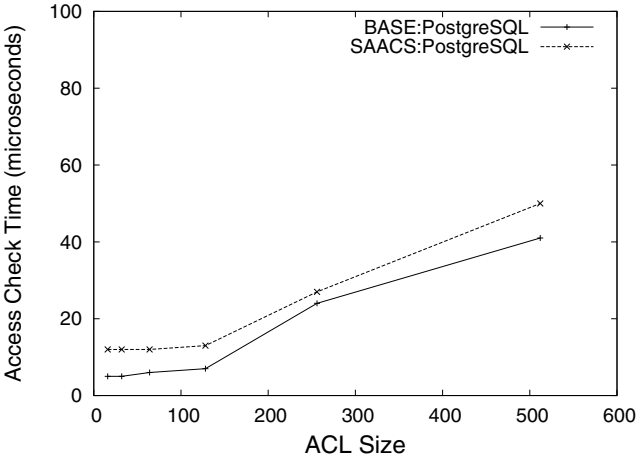
Lising 1. Access Control Enforcement Algorithm in PSAC:PostgreSQL

3.2 Experimental Results

In this section, we report the experimental results comparing the performance of the access control enforcement mechanism in BASE:PostgreSQL and PSAC:PostgreSQL. Specifically, we measure the time required by the access control enforcement mechanism to check the state of a privilege, $test\_priv$, for a user, $test\_user$, on a database table, $test\_table$. We vary the *ACL Size* parameter in our experiments. For BASE:ProstgreSQL, the *ACL Size* is the number of entries in the acl column of the pg_class catalog. For PSAC:PostgreSQL, the ACL size is the combined number of entries in the acl, the $acldeny$, the $aclsuspend$, and the $acltaint$ columns. Note that for the purpose of these experiments we do not maintain any privileges in the *neutral* orientation mode.

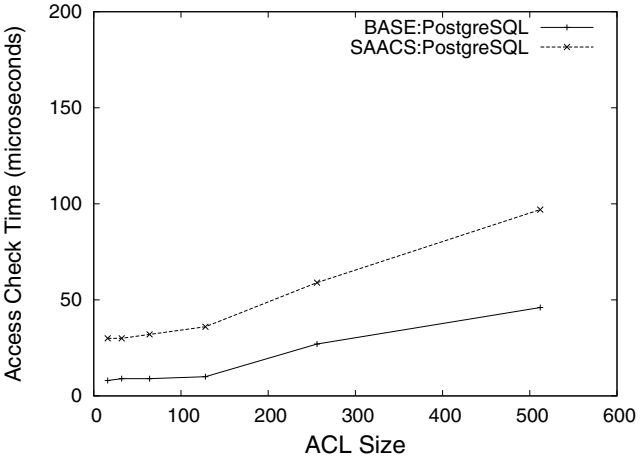
We perform two sets of experiments. The first experiment compares the access control overhead in the absence of a role hierarchy. The results are reported in Figure 2. As expected, the access control overhead for both BASE and PSAC PostgreSQL increases with the ACL Size. The key observation is that the access control overhead for PSAC:PostgreSQL is not much higher than that of BASE:PostgreSQL.

The second experiment compares the access control overhead in the presence of a hypothetically large role hierarchy. We use a role hierarchy of 781 roles with depth equal to 4. The edges and cross-links in the role hierarchy are randomly assigned. The rationale behind such set-up is that we want to observe a reasonable amount of overhead in the access control enforcement code. The role hierarchy is maintained in PSAC:PostgreSQL in a manner similar to that in BASE:PostgreSQL, that is, a role r_p is the parent of a role r_c if r_c is assigned to r_p using the GRANT ROLE command. A role and its assigned roles are stored in the $pg\_auth\_members$ catalog [5]. Next, in the experiment, we randomly assigned 10 roles to the $test\_user$. In order to vary the size of the ACL



Lising 2. Exp 1: Access Control Enforcement Time in BASE and PSAC PostgreSQL in the absence of a role hierarchy

on the $test\_table$, we developed a procedure to assign privileges on the $test\_table$ to randomly chosen roles. Figure 3 reports the results of this experiment. First, observe that the access check time in the presence of a role hierarchy is not much higher than that in the absence of a role hierarchy. As mentioned before, this is mainly because we maintain a cache of the roles assigned to a user (directly or indirectly), thus preventing expensive role inheritance tests at the run-time. Second, the access control enforcement algorithm of PSAC:PostgreSQL reported in Section 3.1 is very efficient with a maximum time of approximately 97 microseconds for an ACL of size 512. Also, it is not



Lising 3. Exp 2: Access Control Enforcement Time in BASE and PSAC PostgreSQL in the presence of a role hierarchy

much higher than the maximum access control enforcement time in BASE:PostgreSQL which stands at approximately 46 microseconds.

Overall, the two experiments confirm the extremely low overhead associated with our design in PSAC:PostgreSQL.

4 Related Work

Access control models have been widely researched in the context of DBMSs [10]. To the best of our knowledge, ours is the first solution formally introducing the concept of privilege states in an access control model.

The implementation of the access control mechanism in the Windows operating system [1], and Network File System protocol V4.1 [3] is similar to the semantics of the *taint* privilege state. In such implementation, the security descriptor of a protected resource can contain two types of ACLs: a Discretionary Access Control List (DACL), and a System Access Control List (SACL). A DACL is similar to the traditional ACL in that it identifies the principals that are allowed or denied some actions on a protected resource. A SACL, on other hand, identifies the principals and the type of actions that cause the system to generate a record in the security log. In that sense, a SACL ACL entry is similar to a PSAC ACL entry with *taint* privilege state. Our concept of privilege states, however, is more general as reflected by the semantics of the other states introduced in our work.

The *up,down*, and *neutral* privilege orientations (in terms of privilege inheritance) have been introduced by Jason Crampton [12]. The main purpose for such privilege orientation in [12] is to show how such scheme can be used to derive a role-based model with multi-level secure policies. However, our main purpose for introducing the privilege orientation modes is to control the propagation of privilege states in a role hierarchy.

Much research work has been carried out in the area of network and host based anomaly detection mechanisms [16]. Similarly, much work on intrusion response methods is also in the context of networks and hosts [19,20]. The fine-grained response actions that we support in this work are more suitable in the context of application level anomaly detection systems in which there is an end user interacting with the system. In that context, an approach to re-authenticate users based on their anomalous mouse movements has been proposed in [17]. In addition, many web applications may force a re-authentication (or a second factor of authentication) in cases when the original authenticator has gone stale (for example expired cookies) to prevent cross-site request forgery (CSRF) attacks.

Foo et. al. [13] have also presented a survey of intrusion response systems. However, the survey is specific to distributed systems. Since the focus of our work is on fine-grained response actions in the context of an application level anomaly detection system, most of the techniques described in [13] are not applicable our scenario.

5 Conclusion

In this paper, we have presented the design, formal model and implementation of a privilege state based access control (PSAC) system tailored for a DBMS. The fundamental

design change in PSAC is that a privilege, assigned to a principal on an object, has a state attached to it. We identify five states in which a privilege can exist namely, *unassign*, *grant*, *taint*, *suspend* and *deny*. A privilege state transition to either the *taint* or the *suspend* state acts as a fine-grained response to an anomalous request. We design PSAC to take into account a role hierarchy. We also introduce the concept of privilege orientation to control the propagation of privilege states in a role hierarchy. We have extended the PostgreSQL DBMS with PSAC describing various design issues. The low access control enforcement overhead in PostgreSQL extended with PSAC confirms that out design is very efficient.

References

1. Access control lists in win32 (June 7, 2009), http://msdn.microsoft.com/en-us/library/aa374872VS.85.aspx
2. Incits/iso/iec 9075. sql-99 standard (January 2, 2009), http://webstore.ansi.org/
3. Nfs version 4 minor version 1 (June 7, 2009), http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-minorversion1-29.txt
4. Oracle database security guide 11g release 1 (11.1) (January 2, 2009), http://download.oracle.com/docs/cd/B28359_01/network.111/b28531/toc.htm
5. The postgresql global development group. postgresql 8.3 (June 7, 2009), http://www.postgresql.org/
6. Postgresql global development group. postgresql 8.3 documentation (January 2, 2009), http://www.postgresql.org/docs/8.3/static/sql-grant.html
7. Sql server 2008 books online. identity and access control (database engine) (January 2, 2009), http://msdn.microsoft.com/en-us/library/bb510418(SQL.100).aspx
8. Bertino, E., Kamra, A., Terzi, E., Vakali, A.: Intrusion detection in rbac-administered databases. In: ACSAC, pp. 170–182. IEEE Computer Society, Los Alamitos (2005)
9. Bertino, E., Samarati, P., Jajodia, S.: An extended authorization model for relational databases. IEEE Transactions on Knowledge and Data Engineering 9(1), 85–101 (1997)
10. Bertino, E., Sandhu, R.: Database security-concepts, approaches, and challenges. IEEE Transactions on Dependable and Secure Computing 2(1), 2–19 (2005)
11. Chandramouli, R., Sandhu, R.: Role based access control features in commercial database management systems. In: National Information Systems Security Conference, pp. 503–511
12. Crampton, J.: Understanding and developing role-based administrative models. In: ACM Conference on Computer and Communications Security, pp. 158–167 (2005)
13. Foo, B., Glause, M., Modelo-Howard, G., Wu, Y.-S., Bagchi, S., Spafford, E.H.: Information Assurance: Dependability and Security in Networked Systems. Morgan Kaufmann, San Francisco (2007)
14. Kamra, A., Bertino, E.: Design and implementation of a intrusion response system for relational database. IEEE Transactions on Knowledge and Data Engineering, TKDE (to appear 2010)
15. Kamra, A., Bertino, E., Terzi, E.: Detecting anomalous access patterns in relational databases. The International Journal on Very Large Data Bases, VLDB (2008)
16. Patcha, A., Park, J.-M.: An overview of anomaly detection techniques: Existing solutions and latest technological trends. Computer Networks 51(12), 3448–3470 (2007)

17. Pusara, M., Brodley, C.E.: User re-authentication via mouse movements. In: ACM Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC), pp. 1–8. ACM, New York (2004)
18. Sandhu, R., Ferraiolo, D., Kuhn, R.: The nist model for role-based access control: Towards a unified standard. In: ACM Workshop on Role-based Access Control, pp. 47–63 (2000)
19. Somayaji, A., Forrest, S.: Automated response using system-call delays. In: Proceedings of the 9th USENIX Security Symposium, p. 185. USENIX Association, Berkeley (2000)
20. Toth, T., Krügel, C.: Evaluating the impact of automated intrusion response mechanisms, pp. 301–310. IEEE Computer Society, Los Alamitos (2002)

Abusing Social Networks for Automated User Profiling

Marco Balduzzi[1], Christian Platzer[2], Thorsten Holz[2],
Engin Kirda[1], Davide Balzarotti[1], and Christopher Kruegel[3]

[1] Institute Eurecom, Sophia Antipolis
[2] Secure Systems Lab, Technical University of Vienna
[3] University of California, Santa Barbara

Abstract. Recently, social networks such as Facebook have experienced a huge surge in popularity. The amount of personal information stored on these sites calls for appropriate security precautions to protect this data.

In this paper, we describe how we are able to take advantage of a common weakness, namely the fact that an attacker can query popular social networks for registered e-mail addresses on a large scale. Starting with a list of about 10.4 million email addresses, we were able to automatically identify more than 1.2 million user profiles associated with these addresses. By automatically crawling and correlating these profiles, we collect detailed personal information about each user, which we use for automated profiling (i.e., to enrich the information available from each user). Having access to such information would allow an attacker to launch sophisticated, targeted attacks, or to improve the efficiency of spam campaigns. We have contacted the most popular providers, who acknowledged the threat and are currently implementing our proposed countermeasures. Facebook and XING, in particular, have recently fixed the problem.

1 Introduction

With the introduction of social networks such as Facebook, the Internet community experienced a revolution in its communication habits. What initially began as a simple frame for social contacts quickly evolved into massively-used platforms where networking and messaging is only one of the multiple possibilities the users can call upon. While basic messaging is still one of the key features, it is clear that the participants see the main advantage in the well-organized representation of friends and acquaintances.

For such an organization to work properly, it is imperative to have certain knowledge about the participants. Suggesting users from the same area with the same age, for instance, can lead to a renewed childhood friendship, while a detailed work history might open unexpected business opportunities. On the other hand, this kind of information is also of great value to entities with potentially malicious intentions. Hence, it is the responsibility of the service provider to ensure that unauthorized access to sensitive profile information is properly restricted. In fact, various researchers (e.g., [1,2,3]) have shown that social networks can pose a significant threat to users' privacy. The main problem is twofold:

– Many users tend to be overly revealing when publishing personal information. Although it lies in the responsibility of each individual to assess the risk of publishing sensitive information, the provider can help by setting defaults that restrict

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 422–441, 2010.

the access to this information to a limited number of individuals. A good example is Facebook, where detailed information is only exchanged between already connected users.
- Information exists in social networks that a user cannot directly control, and may not even be aware of. The best example is the use of the information provided during the registration phase (e.g., name, contact e-mail address, and birthday). Even though this data may never be shown in the public user profile, what most users do not realize is the fact that this information is still often used to provide other functionality within the social network (e.g., such as determining which users might know each other).

In this paper, we describe a novel, practical attack that impacts thousands of users. Moreover, we have shown that this attack is effective against eight of the most popular social networks: Facebook, MySpace, Twitter, LinkedIn, Friendster, Badoo, Netlog, and XING. We discovered that all of these social networks share a common weakness, which is inherent in a feature that is particularly useful for newly-registered users: *Finding friends*. With the functionality to search for friends, social networks need to walk the thin line between revealing only limited information about their users, and simplifying the process of finding existing friends by disclosing the personal details of registered users. A common functionality among these popular social networks is to let users search for friends by providing their e-mail addresses. For example, by entering "gerhard@gmail.com", a user can check if her friend Gerhard has an account on the social network so that she can contact and add him to her friend list. Note that an e-mail address, by default, is *considered to be private information*, and social networks take measures *not to reveal this information*. That is, one cannot typically access a user's profile and simply gain access to his personal e-mail address. One of the main purposes of protecting e-mail addresses is to prevent spammers from crawling the network and collecting e-mail to user mappings. With these mappings at hand, the attacker could easily construct targeted spam and phishing e-mails (e.g., using real names, names of friends, and other personal information [4]). This kind of profiling is also interesting for an attacker to perform a reconnaissance prior to attacking a company. By correlating mappings from different social networks, it is even possible to identify contradictions and untruthfully entered information among profiles.

In our experiments, we used about 10.4 million real-world e-mail addresses that were left by attackers on a dropzone on a compromised machine (which was taken down). We built a system to automatically query each social networking site with these addresses, just as an adversary would, and we were able to identify around 876,000 of these addresses on at least one of the investigated social networks. Furthermore, we implemented a simple guesser that we used to create new e-mail addresses (e.g., for John Doe, addresses such as *john.doe@gmail.com*, *john@gmail.com*, *jdoe@yahoo.com*, etc. would be created) and show that this is an effective and simple technique in practice to find thousands of more accounts.

In summary, we make the following three contributions:

- We describe a real-world, common weakness in eight popular social networks consisting of millions of users, and present a system that automatically takes advantage of this weakness on a large-scale.

- By using e-mail addresses as a unique identifier, we demonstrate that it is possible to correlate the information provided by thousands of users in different social networks. This is a significant privacy threat, because it allows to link profiles that otherwise have no common information. Furthermore, adversaries can leverage this information for sophisticated attacks.
- We present our findings and propose mitigation techniques to secure social networks against such attacks. Our findings were confirmed by all social network providers we contacted. Some of them have already addressed the problem.

The remainder of the paper is structured as follows: In Section 2, we briefly discuss ethical and legal considerations. In Section 3, we explain our attack and how we implemented it for the social networks under examination. In Section 4, we present our findings and assess the potential threat to social networking users. Section 5 discusses possible mitigation solutions. In Section 6, we present related work, with a special focus on privacy-related issues in social networks. We conclude our paper in Section 7.

2 Ethical and Legal Considerations

Crawling and correlating data in social networks is an ethically sensitive area. Similar to the experiments conducted by Jakobsson et al. in [5,6], we believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real-world. Nevertheless, our experiments were designed to protect the users' privacy.

First, for the crawling and correlation experiments we conducted, we only accessed user information that was publicly available within the social networks. Thus, we never broke into any accounts, passwords, or accessed any otherwise protected area or information. Second, the crawler that we developed was not powerful enough to influence the performance of any social network we investigated. Third, we used MD5 on the real names of users to anonymize them properly and handled this data carefully.

We also consulted the legal department of our university (comparable to the IRB in the US), and received a legal statement confirming that our privacy precautions were deemed appropriate and consistent with the European legal position.

3 Abusing E-Mail Querying

Many social network providers such as Facebook, MySpace, XING, or LinkedIn offer a feature that allows a user to search for her friends by providing a list of e-mail addresses. In return, she receives a list of accounts that are registered with these e-mail addresses. From a user's point of view, this feature is valuable: A user can simply upload her address book, and the social network tells her which of her friends are already registered on the site. The feature enables a user to quickly identify other users she knows, and with which she might be interested in establishing a connection.

While the e-mail search functionality commonly available in social networks is convenient, a closer examination reveals that it also has some security-relevant drawbacks. We show that an attacker can misuse this feature by repeatedly querying a large number of e-mail addresses using the search interface as an oracle to validate users on the social network. This information can then be abused in many ways, for example:

- A spammer can automatically validate his list of e-mail addresses (e.g., find out which addresses are most probably real and active) by querying a social network, and only send spam e-mails to those users [7].
- The previous attack can be combined with *social phishing*, i.e., the spammer crawls the profile of a user and uses this information to send targeted phishing e-mails (if the user has a public profile and a public friend list) [4].
- An attacker can generate detailed profiles of the employees of a company and use this information during the reconnaissance phase prior to the actual attack.

Note that it has been recently reported that spammers have started to shift their attention to social networking sites to collect information about users that they can then use for targeted e-mails [8]. The report states that spammers have been using bots to spy information from social networks that they can then use for launching attacks such as guessing passwords (i.e., using reminder hints such as "What is my favorite pet?"). The prerequisite for these current attacks, however, is that a bot is installed on the victim's machine. In comparison, we describe the exploitation of a common weakness in a social network functionality that allows us to retrieve information about users even if they are not infected by a bot.

In each of these cases, the attack is only feasible since the social network provider enables a large-scale query of e-mail addresses. Before going into details on how this feature can be abused in practice, we provide an overview of the context of this type of attacks and previous instances of similar problems.

3.1 Historical Context

Historically, a user search/verification feature was available in many different protocols and services, as we discuss in this section.

SMTP. The *Simple Mail Transfer Protocol* (SMTP) provides two commands, `VRFY` and `EXPN`, to verify a user name or to obtain the content of a mailing list, respectively [9]. A `VRFY` request asks the mail server to verify a given e-mail address, and if a normal response is returned, it must include the mailbox of the user. In addition, an `EXPN` request asks the server for the membership in a mailing list, and a successful response must return the mailboxes on the mailing list.

Spammers began to abuse these two commands to query mail servers for a list of valid e-mail addresses, and to verify if a given e-mail address was in use. Due to this abuse by spammers, SMTP servers now commonly do not provide these two commands anymore (at least not to unauthenticated users).

Finger User Information Protocol. This protocol is used to query a remote server for status and user information [10]. The *finger daemon* typically returns information such as the full name, whether a user is currently logged-on, e-mail address, or similar data. While providing this kind of information is useful in general, an attacker can collect information about a specific user based on the finger protocol, and then use this information for social engineering attacks. Furthermore, the public exposure of the information is questionable from a privacy and security point of view. For these reasons, the majority of Internet hosts does not offer the finger service anymore.

Secure Shell. Certain versions of the OpenSSH server allowed a remote attacker to identify valid users via a timing attack: By analyzing the response time during authentication, an attacker could determine whether or not the supplied username is valid [11]. By adjusting the timing for both successful and failed user verification, this flaw was fixed. A similar flaw can be used to reveal private information with the help of timing attacks against web applications [12].

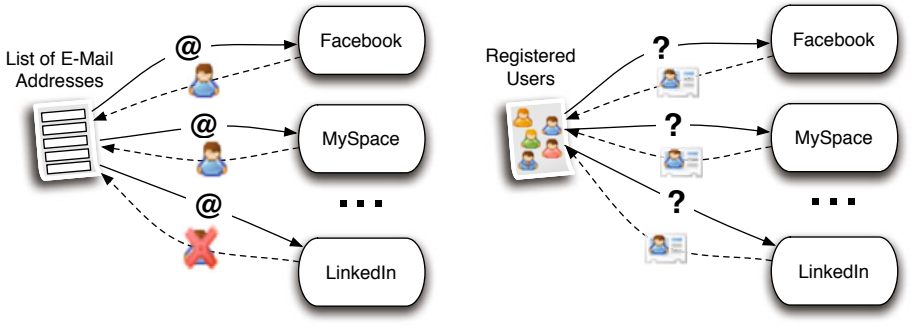
Note that, as discussed above, the conceptual problem that we address in this paper is not necessarily new, but its implications are novel and are far greater because of the large amount of sensitive information contained in user profiles on social networks. We believe that history is repeating itself and that it is only a matter of time before attackers start making use of such features for malicious purposes.

3.2 Automated Profiling of Users

As explained previously, a user can typically send a list of e-mail addresses to a social network and, in return, she receives a mapping of which of these e-mail addresses have a corresponding account on the site. An attacker can abuse this and query for a large number of e-mail addresses on many different social networks (see Figure 1a). As a result, she learns on which social networks the specific address is registered.

In the second step, the attacker retrieves the user's profile from the different networks in an automated way (see Figure 1b). From each profile, she extracts the (publicly-accessible) information she is interested in, for example, age, location, job/company, list of friends, education, or any other information that is publicly available. This information can then be aggregated and correlated to build a rich user profile.

Throughout the rest of this paper, we show that the two steps can indeed be automated to a high degree. Furthermore, we demonstrate that this attack is possible with only very limited resources. In fact, by using a single machine over a few weeks only, we collected hundreds of thousands of user profiles, and queried for millions of e-mail addresses (i.e., each social network was successfully queried for 10.4 million addresses,



(a) Querying social networks for registered e-mail addresses on a large scale.

(b) Crawling every profile found in the first step to collect personal information.

Fig. 1. Automated user profiling based on information collected on social networks

adding up to a total of about 82.3 million queries). This emphasizes the magnitude and the significance of the attack since a more powerful, sophisticated, and determined attacker could potentially extract even more information (e.g., by using a large botnet).

An attacker can also abuse the search feature in a completely different way, extending the attack presented in the previous section. During the profiling step, an attacker can learn the names of a user's friends. This information is often available publicly, including social networking sites such as Facebook and Twitter. An attacker can thus obtain the tuple (first name, last name) for each friend of a given user, but not the e-mail addresses for these friends: The e-mail address itself is considered private information and not directly revealed by the social networking sites. However, an attacker can automatically try to guess the e-mail addresses of the friends of a user by abusing the search feature. We implemented two different, straight-forward techniques for generating new e-mail addresses, based on user names.

For the first technique, for each friend, we build 24 addresses. Given a name in the form *"claudio bianchi"*, we generate six prefixes as *"claudio.bianchi," "claudiobianchi," "claudio_bianchi," "c.bianchi," "c_bianchi,"* and *"cbianchi"*. Then, we append the four most popular free e-mail domains *"gmail.com," "yahoo.com," "aol.com,"* and *"hotmail.com."*

For the second technique, we use context information for generating e-mail addresses: If a user has an e-mail address with a certain structure (e.g., automatically generated e-mail accounts often include the last name of the user and a static prefix), we try to detect this structure by searching the user's first and last name within the e-mail address. If we identify a pattern in the address, we use this match and generate two additional e-mail addresses that follow the same pattern (including both the first and last name) for each friend. If we do not detect a pattern, we generate e-mail addresses similar to the first algorithm. However, instead of appending common prefixes, we use the prefix of the user on the assumption that the friends of a user might be a member of the same e-mail provider.

3.3 Implementation of the Attack

Our prototype system has been implemented as a collection of several components. One component queries the social networks, one extracts and stores the identified information from user profiles, and one automatically correlates the information to discover as much information as possible about a user. An overview of the system and the relationship of the components is shown in Figure 2.

We designed our system to be efficient and stealthy at the same time. Therefore, we had to find a compromise between normal user behavior, which is stealthy, and brute-force crawling, which is efficient but bears the danger of frequently-suspended accounts. Our solution was tweaked for each social network, to find the right combination of timeouts and number of requests. Furthermore, our solutions was carefully designed not to overwhelm the tested networks.

In the following, we describe the system and its components in more detail.

Address Prober. The *Address Prober* is an HTTP client that is responsible for uploading the list of e-mail addresses to be queried to the social network. The social network,

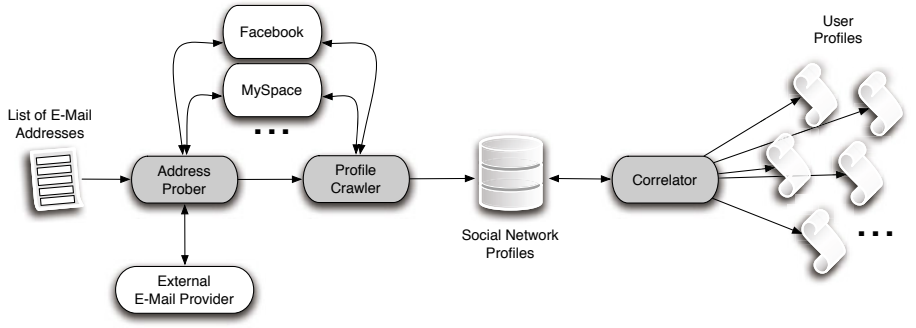


Fig. 2. Overview of system architecture

in return, sends back the list of accounts that are registered with those addresses. The data we are interested in is the profile ID and, if possible, the name, which is attached to the source e-mail address. At this point, some of the processed networks offer some additional data, such as the location or a brief job description.

The client itself is implemented in Python, and embeds an extension to the standard `urllib` library [13] that supports postings in the multipart/form-data format. We adopted such encoding to be able to send to the social networking site a file containing the list of addresses. Typically, this file is required to be formatted in the standard CSV format. On some other networks, for example in Badoo, the list of emails need to be passed as a string of comma-separated addresses.

The Address Prober also supports external e-mail providers such as, for example, Google's webmail service Gmail, and permits to upload lists of e-mail addresses to such accounts. The motivation behind this feature is that some social networks only support e-mail queries if the source is an external e-mail account with an attached address book. Hence, we automatically upload (and afterwards delete again) contacts from specific webmail accounts, before querying the social network.

With this technique, in the worst case (i.e., some sites such as Facebook allow lookups of up to 5,000 addresses), we are able to check sets of around 1,000 e-mail addresses at once. With a short delay, which we set to 30 seconds to ensure that all data is correctly processed and not to overwhelm the network, the prober is able to process data with an overall average speed of 500,000 e-mail addresses per day. A positive side-effect of this technique is that we can query social networks that support Gmail imports in parallel, resulting in a higher overall throughput.

Profile Crawler. The *Profile Crawler* is responsible for a deeper investigation of the user profiles discovered in the previous step. The goal is to gather as much information about a single user as possible. For this purpose, it is mandatory to implement tailored solutions for each supported social network. In the first round, the crawler visits iteratively the user's profile pages for all the social networks, and stores them in a database. On average, we were able to visit 50,000 pages in a single day from a single machine with a single IP address. Some networking sites provided mechanisms to limit the number of profiles visited per day from a single account, while others did not have any limitation mechanism in place. Finally, the crawler automatically parses the data

that has been retrieved and extracts the information of interest, such as sex, age, location, job, and sexual preferences. That is, the Profile Crawler enriches the user profiles discovered by the Address Prober with a set of general and sensitive information.

Correlator. After the crawling phase, the *Correlator* component combines and correlates the profiles that have been collected from the different social networks. The goal of the Correlator is to use the email address as a unique identifier to combine together different profiles and identify the ones that belong to the same person.

When it finds two profiles associated with the same e-mail address, the Correlator compares all the information in the two profiles to identify possible inconsistencies. In particular, it compares all the fields that can assume a small set of values, e.g., sex (either male or female), age (a positive integer number), and current relationship (married, single, or in a relationship).

Using the Correlator, it is possible to automatically infer information that the user might have wanted to keep private. In particular, the correlator has two main goals:

- *Identity Discovery* - If a person provides his full name in social network A, but registers a profile in the network B using a pseudonym, by cross-correlating the two profiles, we can automatically associate the real user's name also to the account B. We are even able to correlate the information about a given person that uses two different pseudonyms by linking the two accounts with the help of the provided e-mail address, which is not possible with the technique proposed by Irani et al. [14]. The combination of information from different sources can be even more worrisome if this information is privacy-relevant. For example, Alice could have a business profile on LinkedIn, and another profile on a dating site in which she does not reveal her real name, but she provides other private information such as her sexual preferences. It is very likely that Alice assumed that it was not possible to link the two "identities" together because there is no public information on the two networks that can be used to match the profiles.
- *Detection of Inconsistent Values* - Sometimes, the information extracted from different social networks is contradictory. For example, Bob can provide his real age on his profile on social network A, while pretending to be 10 years younger on social network B. Again, we can identify this kind of fraudulent (or embellished) profiles in an automated way by cross-correlating the information extracted during crawling the different networks.

4 Evaluation with Real-World Experiments

We performed several experiments on different social networks. As a starting point, we used a set of 10,427,982 e-mail addresses, which were left on a dropzone on a compromised machine that was taken down by law enforcement officials. Based on the log files of this machines, we saw that these e-mail addresses had been used for spamming, and thus, they provided a real-world test case for our system.

4.1 Results for E-Mail Queries

We used our *Address Prober* component on eight social networks, namely Facebook, MySpace, Twitter, LinkedIn, Friendster, Badoo, Netlog, and XING. These networks were chosen because they provide a representative mix of different types of social networks (e.g., business, friendship, and dating). Furthermore, all of these sites have millions of registered users and operate in different countries. Of course, they also vary in their popularity. Facebook, for example, is the most popular social networking site and reports to have more than 400 million active users [15].

Table 1. Discovered profiles

	Network	Query method	E-mail list length	# queried e-mails	# identified	Percentage
		method	*size efficiency*	*speed efficiency*	accounts	
1	Facebook	Direct	5000	10M/day	517,747	4.96%
2	MySpace	GMail	1000	500K/day	209,627	2.01%
3	Twitter	GMail	1000	500K/day	124,398	1.19%
4	LinkedIn	Direct	5000	9M/day	246,093	2.36%
5	Friendster	GMail	1000	400K/day	42,236	0.41%
6	Badoo	Direct	1000	5M/day	12,689	0.12%
7	Netlog	GMail	1000	800K/day	69,971	0.67%
8	XING	Direct	500	3.5M/day	5,883	0.06%
				Total of	1,228,644	11.78%

Table 1 shows the profiles that have been discovered by the e-mail queries that we performed on these sites. Clearly, direct queries to the social networking sites yield faster results than those that are coupled with GMail accounts. Also, while we were able to query 5,000 e-mail addresses at once on Facebook, the best results for XING were 500 addresses per query. The scan method and e-mail list length directly affect the speed of the queries. In general, direct queries are about one order of magnitude faster, and we can check several million e-mail addresses per day. For social networks on which we need to use the GMail support, we can still probe several hundred thousand addresses per day. Also, note that we only adopted a single machine in our tests, while an attacker could perform such an attack in parallel using many machines. In total, we were able to identify 1,228,644 profiles that are linked to one of the e-mail addresses we probed. Most profiles were found on Facebook (4.96%), LinkedIn (2.36%), and MySpace (2.01%).

Table 2 shows the number of profiles that were created with the same e-mail address on different networks. For example, one can see that there are almost 200,000 users who were registered in at least two social networks. In sum, a total of 876,941 unique e-mail addresses we had access to were covered by one or more of the listed social networks.

Table 3 shows the top ten combinations among social networks. That is, the table shows which combinations of networks we typically encountered when we identified a user who is registered on different sites with the same e-mail address. The two most popular combinations are Facebook with MySpace, and Facebook with LinkedIn. Note that the more diverse information a social networking site offers about users as public information, the more significant our attack becomes. In the case of LinkedIn and

Table 2. Overlap for profiles between different networks

Number of Social Networks	Number of Profiles
1	608,989
2	199,161
3	55,660
4	11,483
5	1,478
6	159
7	11
8	0
Total unique	876,941

Table 3. Top ten combinations

Combination	Occurrences
Facebook - MySpace	57,696
Facebook - LinkedIn	49,613
Facebook - Twitter	25,759
Facebook - MySpace - Twitter	13,754
Facebook - LinkedIn - Twitter	13,733
Facebook - NetLOG	12,600
Badoo - FriendSter	11,299
Facebook - MySpace - LinkedIn	9,720
LinkedIn - Twitter	8,802
MySpace - Twitter	7,593

Facebook, we have two social networking sites with different goals. Whereas Facebook aims to foster friendship, LinkedIn aims to foster business collaborations. Hence, we can combine the business information about a user with the more personal, private information they may provide on the friendship site (e.g., under a nickname).

These results of our experiment clearly demonstrates that a potential attacker can easily abuse social networks to enrich his spamming list with the information retrieved from different networks.

4.2 Extracted Information from Profiles

In this section, we provide statistics about the information collected when the *Profile Crawler* visited the user profiles. We present for each of the social networks an overview of what kind of information is available, and also for what percentage of users we were able to extract this information.

Table 4 provides an overview of *general information* such as profile photo, location, and friends available on the different networks. The column *profiles are open* shows the percentage of how many profiles the crawler was able to access, and validate against the name and surname already extracted from the Address Prober. Profiles that are closed include profiles that are configured to be completely private, or that are not accessible

Table 4. Crawling results (values are in percentage): general information

Network	Name Surname	Profiles are open	Photo	Location	Friends	Average friends	Last login	Profile visitors
Facebook	✓	99.89	76.40	0.48	81.98	142	n/a	n/a
MySpace	✓	96.26	55.29	63.59	76.50	137	94.87	n/a
Twitter	✓	99.97	47.59	32.84	78.22	65	n/a	n/a
LinkedIn	✓	96.79	11.80	96.79	96.75	37	n/a	n/a
Friendster	✓	99.72	47.76	99.51	50.23	37	8.79	n/a
Badoo	✓	98.61	70.86	95.23	n/a	n/a	92.01	n/a
Netlog	✓	99.98	43.40	77.54	64.87	31	n/a	73.33
XING	✓	99.88	57.20	96.04	47.25	3	n/a	96.83

Table 5. Crawling results (values are in percentage): sensitive information

	Age	Sex	Spoken language	Job	Education	Current relation	Searched relation	Sexual preference
Facebook	0.35	0.50	n/a	0.23	0.23	0.44	0.31	0.22
MySpace	82.20	64.87	n/a	3.08	2.72	8.41	4.20	4.07
Twitter	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
LinkedIn	n/a	n/a	n/a	96.79	60.68	0.00	n/a	n/a
Friendster	82.97	87.45	n/a	30.88	2.72	64.59	77.76	n/a
Badoo	98.61	98.61	47.81	17.06	19.92	22.48	n/a	22.80
Netlog	97.66	99.99	44.56	43.40	1.64	25.73	23.14	29.30
XING	n/a	n/a	84.54	99.87	49.21	n/a	n/a	n/a

anymore. In Facebook, more than 99% of the profiles are open, but only little information is shown by default to anonymous users and persons that are not a friend of the user. On the contrary, the profile photo and the list of friends are usually accessible.

Typically, the different pieces of information can be either private or public, and the social network provider assigns a default value for the privacy setting of them. From our observations, it seems that many users do not change the default privacy settings for a specific type of information. Note that when some data is not accessible, it either means that the user has not set such a value (it is optional) or that is not accessible due to privacy reasons.

Table 5 shows the availability of *sensitive information* on the individual networks. Depending on the purpose of the social network, different types of information are made public by default. Dating platforms, for instance, focus on personal information such as age, sex, or current relationship status, while business networks emphasize educational and work-related aspects. However, all of these details are more sensitive and can be used for the accurate profiling of an account. Precise values such age and sex can easily be correlated across different social networks, essentially forming richer sets of publicly available data than initially desired by the user. We provide a detailed overview of this aspect in Section 4.4.

Table 6. Crawling results: extra information

Network	Personal homepage	Phone	Birthday	IMs	Physical appearance	Income	Prof. skills	Interests Hobbies
Facebook	✓	✓	✓	✓			✓	✓
MySpace			✓		✓	✓		✓
Twitter	✓						✓	✓
LinkedIn	✓	✓	✓	✓			✓	✓
Friendster								✓
Badoo	✓			✓	✓			✓
Netlog						✓		✓
XING	✓	✓	✓	✓			✓	✓

Finally, Table 6 shows what kind of *additional information* each social network supports. We refrain from providing a percentage for these fields, because this type of information is only available for a minority of the sampled cases.

4.3 Automated Guessing of User Profiles

While it is useful for the attacker to have access to e-mail lists that she can readily query, it is also interesting for her to automatically generate new e-mail addresses that she could then re-validate against the social networks. Using the *e-mail guesser* as discussed earlier, we are able to generate addresses that we do not previously know, and verify their existence in social networks. By starting with 650 profiles and using straightforward automated e-mail guessing techniques, we were able to identify the e-mails of about 20,000 users along with their associated profiles (a thirty-fold increase compared to the initial profile set). Hence, our experiment demonstrated that even if the attacker does not have access to a large e-mail database, by starting with a small set, she can still successfully guess addresses and identify thousands of new profiles.

4.4 Detecting Anomalous Profiles by Cross-Correlation

In the following, we present the output of the correlation phase, and we discuss several interesting examples of anomalous profiles we automatically discovered during our empirical experiments.

Discovering Mismatched Profiles. Based on the data provided by the different social networks, we configured the Correlator to analyze six information fields that are popular among the different social networks we examined: Name, location, age, sex, current relationship, and sexual preference.

Before proceeding to the comparison, we had to normalize the values provided in the different social networks to a common dictionary. For example, sex was translated to either “male” or “female,” while the current relationship and the sexual preference’s values were translated into a set of common keywords built from an analysis of the entire dataset. For instance, values like “heterosexual,” “straight,” and “man looking for women” were all translated into the keyword “heterosexual.” Likewise, we normalized

Table 7. Information provided on multiple profiles belonging to the same user

	# of Occurrences on X networks						
Information	2	3	4	5	6	7	Total
Name	199,161	55,660	11,483	1,478	159	11	267,952
Location	22,583	2,102	174	11	3		24,873
Age	19,135	887	36				20,085
Sex	17,282	854	34				18,170
Sexual preference	760	13					773
Current relation	1,652	38	1				1,691

the current relationship field to one of the four following values: "Single," "in a relationship," "married," and "complicated." Finally, we filtered the geographical location by comparing the field (or part of it) against a dictionary of more than 260,000 cities.

Table 7 shows the number of users that provide a certain information on multiple social networks. For example, 22,583 users included their location on two networks, 2,102 on three networks, and 174 on four networks. Since the name is a mandatory field, the first line of the table matches the number of profiles reported in Table 2.

For each field, the Correlator computed the total number of distinct values provided by the same users across different networks. For example, if Alice is registered on three social networks where she provides as age 26, 26, and 22 the Correlator reports two mismatched values.

Table 8. Overview of profiles where a mismatch was detected - Data are normalized.

		% Total	% of mismatched values		
Information	Value	mismatches	2	3	4+
Name	*string*	72.65	62.70	35.37	17.66
Location	*city*	53.27	51.74	16.24	3.72
Age	$0 < n < 100$	34.49	33.58	17.84	30.56
Sex	*m, f*	12.18	12.18		
Sexual preference	*hetero, homo, bi*	7.63	7.63		
Current relation	*single, relationship, married, complicated*	35.54	35.42	5.13	

Table 8 summarizes the results. The first column shows the percentage of profiles, from the total shown in Table 7, for which the Correlator found mismatching values. About one-third of the people seems to misrepresent their current relationship status, declaring, for example, to be single on one network and to be married on a second one. It is also interesting to note that 2,213 users (12% of the ones registered in more than one network) pretend to be male on a network and female on a different one. The very high percentage of people using different names is a consequence of various factors. First, the name comparison is more problematic because, as explained in Section 2, we only store the MD5 of the names to preserve the users privacy. This means that we lose the ability to distinguish between small differences and completely fake names. Second, in some social networks, it is very common to provide a nickname instead of

the real user name. For example, John Doe on LinkedIn may appear simply as JDoe77 on MySpace.

The last three columns in Table 8 show how many unique values where provided by the same user (either two, three, or more) on different social networks. These percentages are normalized by the number of accounts owned by the user. That is, a value of 10% in Column 3 means that 10% of the people that own an account on at least three social networks provided three different values for that specific field.

Mismatches in Provided Age Information. Five of the eight social networks we examined either offer the possibility for a user to specify his age, or automatically compute this information from the user's birthday. During our experiments, the *Correlator* automatically found a total of more than 20,000 users for which we had at least two profiles on different networks which also included the person's age. Surprisingly, about one-third of these users (6,919) showed a mismatch for the age information provided in the different profiles (see Table 9 for details). This number only includes those profiles in which the difference of age is at least two years. In fact, a mismatch of only one year is likely to be the consequence of outdated profiles (i.e., if a user needs to manually specify his age, he might forget to update the value at each birthday).

Table 9. Overview of profiles where a mismatch was detected in the age.

Range	#	%
2 - 10	4,163	60.17
11 - 30	1,790	25.87
31 +	966	13.96
Profiles with Age	20,085	
Total mismatched	6,919	

Among the profiles with an age that differs more than two years, we were able to identify 712 users (10% of this set) who claim to be underage, even though they appear to be more than 18 years old in another networks (or vice versa). For example, we observed that many teenagers increase their age to register to Badoo, since the site restricts its access to adults only.

A Short Glimpse into Hidden Profiles. Probably the most serious consequence of the attack presented in this paper is the ability to uncover hidden relationships between different profiles, allowing an attacker to infer private information about a subject.

By looking at the results of our experiments, it is possible to find examples of possibly hidden profiles and online identities that users probably wish to keep secret. As a proof of concept of the impact that correlating profile information can have on a user's privacy, we picked some random profiles that showed mismatching values. In one case, a married person owned an account on a dating-related social network under a different name, with a list of friends who were much younger. While such information may be a complete misinterpretation, nevertheless, there may be many cases where an attacker may try to use the information to his advantage.

Because of the ethically sensitive aspects of analyzing this kind of interconnections, we did not perform an in-depth investigation of the problem, limiting the result of our analysis to aggregated figures.

5 Countermeasures

In this section, we present several mitigation strategies that can be used to limit the extent of our attack. Each approach has its own advantages and limitations, which we review in the rest of the section. We discussed the different countermeasures with several popular social network providers to incorporate also their view of the problem, especially considering the operational practicability of each proposed solution.

1) Raising Awareness: Mitigation From the User's Perspective. Clearly, if users were to use a different e-mail address on each social networking site, it would become more difficult for the attacker to automatically correlate the extracted information. Because the e-mail address is the unique ID that identifies a specific user, an effective defense technique would be to educate users to use a different e-mail address each time they register for and enter personal information into a social networking site. Unfortunately, educating users on security and privacy issues is not an easy task. Often, users may choose to ignore the risks and opt for the ease of use (e.g., analogous to users using the same password across many web sites – which has been reported to be quite common [16]).

2) Possible Solution: CAPTCHAs. When searching for e-mail addresses, a user could be forced to solve a CAPTCHA (i.e., a type of challenge-response test which is hard to solve for a computer [17]). This would prohibit automated, large-scale queries to a certain extent since CAPTCHAs cannot be (easily) solved by a computer.

However, introducing this kind of countermeasure has three main drawbacks. First, the user experience is reduced if a user needs to solve a CAPTCHA frequently, and this should be avoided by all means. Even if solving a CAPTCHA is only required for users that perform many queries, the network operators tend to dislike this mitigation strategy due to a potential negative user experience. Second, using this approach is not a real solution to the problem since an attacker can also hire *real* people to solve the challenge-response tests. This type of service is surprisingly cheap on the underground market, with 1,000 solved CAPTCHAs costing about $2 [18]. Third, different CAPTCHA systems are prone to attack such that a computer can solve the test with a reasonable success rate, completely defeating the countermeasure [19,20,21].

3) Possible Solution: Contextual Information. Another potential approach to mitigate the problem is to require *contextual information* for each query. If a user U wishes to search for his friends $F_1, F_2, \ldots F_n$, he has some context information for each of them that he should include in his query. For example, a user knows the full name of each friend, he can estimate their age, or knows their approximate location. It is probable that the attacker lacks this kind of information.

Unfortunately, it is inconvenient for a user to provide contextual information to perform a query. While a user can, for example, store the full name together with the e-mail

address within the address book application, this information might no be available for all friends. Furthermore, additional contextual information such as age or location needs to be provided manually. As a result, this solution is likely not feasible from an operational point of view.

4) Possible Solution: Limiting Information Exposure. Our attack is possible since the search result contains a mapping between the queried e-mail address and the profile name (if an account with this e-mail address is registered). Thus, a viable option to prevent our attack is to not return a mapping between e-mail address and profile name in the search result. This could, for example, be implemented by simply returning a list of registered accounts in a random order, without revealing which e-mail address belongs to which account. Note that a user typically does not need the correct mapping, he is only interested in the fact that one of his friends is registered on the social network such that she can add him to his friends list.

5) Possible Solution: Incremental Updates. Another fact that enables our attack is the huge number of searches we can perform: We can query thousands of e-mail addresses at once, and also repeat this process as often as we wish. A natural approach for mitigation is, thus, to implement some kind of limitation for the queries a user is allowed to perform. For example, by enforcing *incremental updates*, a user is allowed to initially query many e-mail addresses, but this step can only be performed once. This enables a user to search for his current friends on the given social network in the beginning. Afterwards, the number of queries can be restricted to only a small number of e-mail addresses (for example only 50). This enables a user to incrementally extend his network, but also limits the number of e-mail addresses a user can search for.

6) Possible Solution: Rate-limiting Queries. Another viable option to limit our attack is *rate-limiting* the number of queries: That is, we restrict the (total) number of queries a user can send to the social network, therefore limiting the amount of e-mail addresses a given user can check. An option could be to either rate-limit the number of queries (e.g., only two large queries per week) or have a total upper bound of e-mail addresses a user can search for (e.g., a total of 10K e-mail addresses a user can check).

Most social network providers already have different kinds of rate-limiting in place. For example, rate-limiting is used to prohibit automated crawling of their site, or regulating how many messages a given user can send per day to stop spamming attacks. Therefore, rate-limiting the number of e-mail searches a user is allowed to perform fits into the operational framework of these sites. When we contacted the most popular social network providers, the majority of them preferred this solution. In the meantime, Facebook and XING have already implemented this countermeasure and now limit the number of lookups that can be performed by a single source.

Limitations of the Countermeasures. Note that although there is much room for improvement in defending against e-mail-to-account mapping information leakage attacks, the attacker could still extract information from the social networking site for specific, targeted users (e.g., by only sending e-mail queries consisting of a single user). Hence, if social networking sites choose to provide e-mail searching functionality, there

is always a potential for misuse and the privacy of the users may be at risk. However, the countermeasures we described in this section raise the difficulty bar for the attacker, mitigating the problem at least on a large scale.

6 Related Work

The large popularity of social networks and the availability of large amounts of personal information has been unprecedented on the Internet. As a result, this increasing popularity has lead to many recent studies that examine the security and privacy aspects of these networks (e.g., [3,4,7,22,23,24,25,26]). As more and more Internet users are registering on social networking sites and are sharing private information, it is important to understand the significance of the risks that are involved.

The structure and topology of different social networks was examined by different research groups (e.g., [27,28,29,30]). The main focus of previous work was either on efficient crawling or on understanding the different aspects of the graph structure of social networks. We extend previous work by contributing a novel way to enumerate users on social networks with the help of e-mail lookups. Furthermore, we implemented several efficient crawlers for social networks and – to the best of our knowledge – we are the first to perform large-scale crawls of eight social networks.

Our attack is facilitated by the fact that an attacker can use an e-mail address to link profiles on different social networks to a single user. The idea of correlating data from different sources to build a user profile has been studied in different contexts before. For example, Griffith and Jakobsson showed that it is possible to correlate information from public records to better guess the mother's maiden name for a person [31]. Heatherly et al. [32], and Zheleva and Getoor [33] recently showed that hidden information on a user's profile can also be inferred with the help of contextual information (e.g., the political affiliation of a user can be predicted by examining political affiliation of friends).

Concurrently and independently of our work, Irani et al. [14] performed a similar study of social networks. They showed that it is straightforward to reconstruct the identify (what they call the *social footprint*) of a person by correlating social network profiles of different networks. The correlation is done either by using the user's pseudonym or by inferring it from the user's real name. In contrast, our work focuses on automated techniques to find profiles of the same person on different networks. In fact, due to the friend-finder weakness that we discovered on all tested networks, we are able to associate profiles by e-mail addresses. As a result, we produce a more precise correlation: On one hand, we can make sure that different profiles belong to the same individual (Irani et al. have a positive score of only 40% for the pseudonym match and 10%-30% for the real name match). On the other hand, we can reveal the "'hidden profiles" of users that they may actually wish to hide. Indeed, this is a major advantage of our approach; we can link profiles that are registered using different pseudonyms or information, but based on the same e-mail address. Finally, we conducted our studies on a larger set of data by crawling 876,941 unique profiles (versus 21,764 profiles studied by Irani et al.) and extracting up to 15 information fields from each profile (versus 7).

Also, note that our work is also related to the area of *de-anonymization*, where an attacker tries to correlate information obtained in different contexts to learn more about

the identity of a victim. Narayanan and Shmatikov showed that by combining data with background knowledge, an attacker is capable of identifying a user [34]. They applied their technique to the Internet movie database (IMDb) as background knowledge and the Netflix prize dataset as an anonymized dataset, and were indeed able to recognizes users. Furthermore, the two researchers applied a similar technique to social networks and showed that the network topology in these networks can be used to re-identify users [35]. Recently, Wondracek et al. [36] introduced a novel technique based on social network groups as well as some traditional browser history-stealing tactics to reveal the actual identity of users. They based their empirical measurements on the XING network, and their analysis suggested that about 42% of the users that use groups can be uniquely identified.

In this paper, we continue this line of work and show that an attacker can cross-correlate information between different social networking sites in an automated way. The collected information reveals the different online identities of a person, sometimes uncovering "secret" profiles.

7 Conclusion

In this paper, we presented a novel attack that automatically exploits a common weakness that is present in many popular social networking sites. We launched real-world experiments on eight distinct social networks that have user bases that consist of millions of users. We leverage the fact that an attacker can query the social network providers for registered e-mail addresses on a very large scale. Starting with a list of about 10.4 million e-mail addresses, we were able to automatically identify more than 1.2 million user profiles associated with these addresses.

We can automatically crawl the user profiles that we map to e-mail addresses, and collect personal information about each user. We then iterate through the extracted friend lists to generate an additional set of candidate email addresses that can then be used to discover new profiles. Our attack is significant because we are able to correlate information about users across many different social networks. That is, users that are registered on multiple social networking web sites with the same e-mail address are vulnerable. Our experiments demonstrate that we are able to automatically extract information about users that they may actually wish to hide certain online behavior. For example, we can identify users who are potentially using a different name on a dating web site, and are pretending to be younger than they really are. The correlation that we are able to do automatically has a significant privacy impact.

After we conducted our experiments and verified the feasibility of our attack, we contacted the most popular social network providers such as Facebook, MySpace, XING and Twitter, who all acknowledged the threat, and informed us that they are going to adopt some of our countermeasures. By now, Facebook and XING have already fixed the problem by limiting the number of requests that a single source can perform, and we expect that other social networks will also implement countermeasures.

Acknowledgments. This work has been supported by Secure Business Austria, by the European Commission through project FP7-ICT-216026-WOMBAT, by the POLE de Competitivite SCS (France) through the MECANOS project and by the French National Research Agency through the VAMPIRE project.

References

1. Dwyer, C., Hiltz, S.: Trust and Privacy Concern Within Social Networking Sites: A Comparison of Facebook and MySpace. In: Proceedings of the Thirteenth Americas Conference on Information Systems, AMCIS (2007)
2. Fogel, J., Nehmad, E.: Internet social network communities: Risk taking, trust, and privacy concerns. Comput. Hum. Behav. 25(1), 153–160 (2009)
3. Gross, R., Acquisti, A., Heinz III, H.J.: Information revelation and privacy in online social networks. In: ACM Workshop on Privacy in the Electronic Society, WPES (2005)
4. Jagatic, T.N., Johnson, N.A., Jakobsson, M., Menczer, F.: Social phishing. ACM Commun. 50(10), 94–100 (2007)
5. Jakobsson, M., Finn, P., Johnson, N.: Why and How to Perform Fraud Experiments. IEEE Security & Privacy 6(2), 66–68 (2008)
6. Jakobsson, M., Ratkiewicz, J.: Designing ethical phishing experiments: a study of (ROT13) rOnl query features. In: 15th International Conference on World Wide Web, WWW (2006)
7. Brown, G., Howe, T., Ihbe, M., Prakash, A., Borders, K.: Social networks and context-aware spam. In: ACM Conference on Computer Supported Cooperative Work, CSCW (2008)
8. News, H.: Spam-Bots werten soziale Netze aus (September 2009), `http://www.heise.de/security/Spam-Bots-werten-soziale-Netze-aus-/news/meldung/145344`
9. Klensin, J.: Simple Mail Transfer Protocol. RFC 5321 (Draft Standard) (October 2008)
10. Zimmerman, D.: The Finger User Information Protocol. RFC 1288 (Draft Standard) (December 1991)
11. Bugtraq: OpenSSH-portable Enabled PAM Delay Information Disclosure Vulnerability (April 2003), `http://www.securityfocus.com/bid/7467`
12. Bortz, A., Boneh, D.: Exposing private information by timing web applications. In: 16th International Conference on World Wide Web (2007)
13. Python Software Foundation: Python 2.6 urllib module, `http://docs.python.org/library/urllib.html`
14. Irani, D., Webb, S., Li, K., Pu, C.: Large online social footprints–an emerging threat. IEEE International Conference on Computational Science and Engineering 3, 271–276 (2009)
15. Facebook: Statistics (April 2010), `http://www.facebook.com/press/info.php?statistics`
16. Florencio, D., Herley, C.: A large-scale study of web password habits. In: 16th International Conference on World Wide Web (WWW), New York, NY, USA (2007)
17. von Ahn, L., Blum, M., Hopper, N.J., Langford, J.: CAPTCHA: Using Hard AI Problems for Security. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656. Springer, Heidelberg (2003)
18. Danchev, D.: Inside India's CAPTCHA solving economy (August 2008), `http://blogs.zdnet.com/security/?p=1835`
19. Chellapilla, K., Simard, P.Y.: Using Machine Learning to Break Visual Human Interaction Proofs (HIPs). In: Neural Information Processing Systems, NIPS (2004)
20. Mori, G., Malik, J.: Recognizing Objects in Adversarial Clutter: Breaking a Visual CAPTCHA. In: IEEE Conference on Computer Vision & Pattern Recognition, CVPR (2003)

21. Yan, J., El Ahmad, A.S.: A low-cost attack on a Microsoft CAPTCHA. In: 15th ACM Conference on Computer and Communications Security, CCS (2008)
22. Bilge, L., Strufe, T., Balzarotti, D., Kirda, E.: All Your Contacts Are Belong to Us: Automated Identity Theft Attacks on Social Networks. In: 18th International Conference on World Wide Web, WWW (2009)
23. Bonneau, J., Preibusch, S.: The Privacy Jungle: On the Market for Privacy in Social Networks. In: Workshop on the Economics of Information Security, WEIS (2009)
24. Chew, M., Balfanz, D., Laurie, B.: (Under)mining Privacy in Social Networks. In: Proceedings of Web 2.0 Security and Privacy Workshop, W2SP (2008)
25. Jones, S., Millermaier, S., Goya-Martinez, M., Schuler, J.: Whose space is MySpace? A content analysis of MySpace profiles. First Monday 12(9) (August 2008)
26. Krishnamurthy, B., Wills, C.E.: Characterizing Privacy in Online Social Networks. In: Workshop on Online Social Networks, WOSN (2008)
27. Bonneau, J., Anderson, J., Danezis, G.: Prying Data out of a Social Network. In: First International Conference on Advances in Social Networks Analysis and Mining (2009)
28. Chau, D.H., Pandit, S., Wang, S., Faloutsos, C.: Parallel Crawling for Online Social Networks. In: 16th International Conference on World Wide Web, WWW (2007)
29. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and Analysis of Online Social Networks. In: ACM SIGCOMM Conference on Internet Measurement, IMC (2007)
30. Wilson, C., Boe, B., Sala, A., Puttaswamy, K.P.N., Zhao, B.Y.: User Interactions in Social Networks and their Implications. In: 4th ACM European Conference on Computer Systems (EuroSys). ACM, New York (2009)
31. Griffith, V., Jakobsson, M.: Messin' with texas, deriving mother's maiden names using public records. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 91–103. Springer, Heidelberg (2005)
32. Raymond Heatherly, M.K., Thuraisingham, B.: Preventing private information inference attacks on social networks. Technical Report UTDCS-03-09, University of Texas at Dallas (2009)
33. Zheleva, E., Getoor, L.: To Join or Not To Join: The Illusion of Privacy in Social Networks with Mixed Public and Private User Profiles. In: 18th International Conference on World Wide Web, WWW (2009)
34. Narayanan, A., Shmatikov, V.: Robust De-anonymization of Large Sparse Datasets. In: IEEE Symposium on Security and Privacy (2008)
35. Narayanan, A., Shmatikov, V.: De-anonymizing social networks. In: IEEE Symposium on Security and Privacy (2009)
36. Wondracek, G., Holz, T., Kirda, E., Kruegel, C.: A Practical Attack to De-Anonymize Social Network Users. In: IEEE Symposium on Security and Privacy (2010)

An Analysis of Rogue AV Campaigns*

Marco Cova[1], Corrado Leita[2], Olivier Thonnard[3], Angelos D. Keromytis[4], and Marc Dacier[2]

[1] University of California Santa Barbara, Santa Barbara, USA
marco@cs.ucsb.edu
[2] Symantec Research Labs, Sophia Antipolis, France
{corrado_leita,marc_dacier}@symantec.com
[3] Royal Military Academy, Brussels, Belgium
olivier.thonnard@rma.ac.be
[4] Columbia University, New York, USA
angelos@cs.columbia.edu

Abstract. Rogue antivirus software has recently received extensive attention, justified by the diffusion and efficacy of its propagation. We present a longitudinal analysis of the rogue antivirus threat ecosystem, focusing on the structure and dynamics of this threat and its economics. To that end, we compiled and mined a large dataset of characteristics of rogue antivirus domains and of the servers that host them.

The contributions of this paper are threefold. Firstly, we offer the first, to our knowledge, broad analysis of the infrastructure underpinning the distribution of rogue security software by tracking 6,500 malicious domains. Secondly, we show how to apply attack attribution methodologies to correlate campaigns likely to be associated to the same individuals or groups. By using these techniques, we identify 127 rogue security software campaigns comprising 4,549 domains. Finally, we contextualize our findings by comparing them to a different threat ecosystem, that of browser exploits. We underline the profound difference in the structure of the two threats, and we investigate the root causes of this difference by analyzing the economic balance of the rogue antivirus ecosystem. We track 372,096 victims over a period of 2 months and we take advantage of this information to retrieve monetization insights. While applied to a specific threat type, the methodology and the lessons learned from this work are of general applicability to develop a better understanding of the threat economies.

* This work has been partially supported by the European Commission through project FP7-ICT-216026-WOMBAT funded by the 7th framework program. The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission. This work was also partly supported by ONR through Grant N00014-07-1-0907 and the NSF through Grant CNS-09-14845. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the ONR or the NSF. The work of Marco Cova was supported by a fellowship made possible by Symantec Research Labs.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 442–463, 2010.

1 Introduction

A rogue security software program is a type of misleading application that pretends to be legitimate security software, such as an anti-virus scanner, but which actually provides the user with little or no protection. In some cases, rogue security software (heretofore referred to as "rogue AV") actually facilitates the installation of the very malicious code that it purports to protect against.

Rogue AVs typically find their way into victim machines in two ways. First, social engineering techniques can be used to convince inexperienced users that a rogue tool is legitimate and that its use is necessary to remediate often nonexistent or exaggerated threats found on the victim's computer. A second, stealthier technique consists of attracting victims to malicious web sites that exploit vulnerabilities in the client software (typically, the browser or one of its plugins) to download and install the rogue programs without any user intervention (*e.g.,* through *drive-by* downloads). After a rogue AV is installed on a victim's machine, it uses a number of techniques to convince (or force) a user to pay for additional tools or services, such as a "full version" of the program or the subscription to an update service. The cost of these additional programs or services ranges from $30–$100 [8].

In the last few years, rogue AVs have become a major security threat, both in terms of their pervasiveness and their financial impact. For example, over a 1-year period, Symantec's sensors detected 43 million installation attempts, covering over 250 distinct families of rogue AV software [8]. In addition, an investigation by Krebs revealed that affiliate programs alone can generate upward of $300,000 a month for the individuals that distribute rogue AVs [14].

As a consequence, different companies in the computer security industry have recently focused their attention on this threat [1,4,8]. Most of the existing works have considered individual facets of the rogue AV problem, for example, the malware code (*e.g.,* the installation techniques it employs), the sites involved in its distribution (*e.g.,* their number and geolocation), and the victims that it affects. However, little has been done to understand the rogue AV phenomenon as a whole, that is, relating how these individual pieces become combined in rogue AV campaigns.

We seek to fill this gap by providing a better understanding of the organization and dynamics of rogue AV campaigns. In particular, we focus on characterizing the infrastructure used in a campaign (*e.g.,* its web servers, DNS servers, and web sites) and the strategies used to create and manage it. We also investigate the uniqueness of our findings to this very specific threat type, and we investigate the motivations underneath these differences by exploring its economics.

The key of our approach is a method that, given a list of individual AV-hosting sites, allows us to group them into campaigns, each characterized by coherent features. More precisely, we use an extensive dataset including network and domain registration information, as well as network-observable temporal characteristics of a large number of domains that are associated with rogue AV advertising and distribution. To that dataset we apply a multi-criteria fusion algorithm to group together nodes based on a certain number of common elements likely due to the

same root cause (*e.g.,* the same rogue AV campaign). This attribution method uses a combination of unsupervised, graph-based clustering combined with a data aggregation method inspired by multi-criteria decision analysis (MCDA). On the one hand, our approach enables the identification of rogue AV campaigns and the analysis of the *modus operandi* of the individuals controlling them. On the other hand, this approach enables the execution of comparative analyses and the assessment of the uniqueness of the findings to the specific threat landscape.

The specific contributions of our work described in this paper include:

- The first, to our knowledge, large-scale analysis of the rogue AV threat and of its distribution infrastructure, conducted by tracking 6,500 malicious domains.
- A demonstration of the usefulness of attack attribution approaches to the problem of mining large security datasets. We show how, by using MCDA, we are able to discover specific campaigns likely to be associated to the action of a specific individual or group.
- The first characterization of the behavior of rogue AV campaigns and their economics. We reveal insights on the automated deployment of large amounts of domains, and we demonstrate their specificity to the threat landscape by comparing the results to those associated with other web-born threats. We collect information on 372,096 users (clients) interacting with some rogue AV domains to generate information on the average conversion rate of a rogue AV campaign. We demonstrate the existence of a very specific economic balance, that justifies a bigger investment in the deployment and maintenance of such large-scale campaigns.

The remainder of this paper is organized as follows. Section 2 describes the state of the art on tracking and mitigating the rogue AV threat. Section 3 describes the features we used in our analysis, as well as the clustering technique itself. Section 4 highlights our most interesting insights following the analysis, while Section 5 assesses the specificity of our findings to the rogue AV threat, and looks into their economic motivations. Finally, Section 6 summarizes some of the lessons we learned from this study, and Section 7 concludes the document.

2 State of the Art

The presence of rogue security software has been observed (at least) as early as 2005 [28]. More in-depth reports of rogue security software have ranged from analyses on the diffusion of such threats [1], to studies on their social aspects and their comparison to traditional malware economies [18]. Recently, security industry reports [4,8] have presented thorough descriptions of various instances of rogue software, their look and feel as well as some tactics they use. By focusing on a large-scale study of the structure of the distribution infrastructure for rogue AV domains, this work complements previous analyses on rogue security software by offering new lessons on this threat ecosystem and its peculiarities.

We previously provided a preliminary, high-level overview of some of the results obtained with the method described in this paper [8]. The novel contributions of this paper with respect to that technical report are threefold. First, we provide a precise description of the experimental setup and the analysis method. Second, we give a comparison, thanks to a novel experimental dataset, with other kinds of web-based threats. Third, we supply an ensemble of insights on the economic rationales explaining the identified differences.

Concurrently to our work, Google published a study on the distribution of rogue AV software [21], which focuses on the prevalence of rogue AV software and on its distribution mechanisms. In this paper, we also uncovered the campaigns underlying rogue AV sites and performed an initial study of their victims.

These economic insights contribute at completing the picture on the underground economy and its dynamics. This complements previous works on the topic. Similarly to what is presented here, Moore *et al.* [16] have collected client volume information for a different threat landscape, that of the phishing websites. Holz et al. [11] have instead infiltrated some weakly configured drop-zones to study the extent and the economic aspects of phishing and attack campaigns. Finally, previous work [7,9] has monitored the type of transactions carried out by cyber-criminals through underground channels.

Different techniques have been proposed to study the structure and the diffusion of specific threats. Moshchuk *et al.* [17] have crawled 18 million URLs to quantify the nature and the extent of the spyware problem. Provos *et al.* [19] have analyzed billions of URLs and used a combination of machine learning techniques and crawlers to analyze the infrastructure underneath drive-by downloads. McGrath *et al.* [15] have studied the dynamics associated to the registration of phishing domains. Stone-Gross *et al.* [24] have infiltrated the Torpig botnet and gathered information on its size and dynamics. In all these cases, the authors have used different data collection techniques to generate high-level overviews on specific threats. While this work complements the state of the art by providing an analysis of a previously unexplored threat landscape, that of the rogue security software, our contribution goes beyond that. We show the usefulness of multi-criteria analysis techniques to mine these large datasets and discover specific campaigns within the multitude of domains under observation. We also demonstrate our ability to leverage these techniques to compare different threat landscapes, and identify specific behaviors that are a characteristic of a given threat.

3 Methodology

In this Section, we begin by describing our methodology for collecting information about the rogue AV distribution infrastructure. We then discuss the analysis techniques that we used on the collected data. The data collection itself was carried out over three separate phases: the collection of rogue AV-related domain names, the collection of information on each domain and the discovery of specific campaigns leveraging attack attribution tools developed in the context of the WOMBAT project[1].

[1] `http://www.wombat-project.eu`

3.1 Rogue AV Domains

To build an initial seed of domains associated to the rogue AV distribution, we aggregated information from a number of different sources:

- Norton Safeweb (`http://safeweb.norton.com`)
- Malware Domain List (`http://malwaredomainlist.com`)
- Malware URL (`http://www.malwareurl.com`)
- Hosts File (`http://www.hosts-file.net`)

All these sources offer at least a rough categorization of the type of each malicious domain they are listing, and allowed us to systematically collect all the domains that were believed to be correlated to the rogue AV distribution by means of simple heuristics.

To complete our picture on the collected domains, we have integrated our domain list with the information generated by freely accessible IP-NAME mapping datasets (`http://www.robtex.com`). This allowed us to discover all the domain names hosted on each IP where at least one rogue domain had been found.

3.2 Rogue Server Information

Once the initial list of domains was created, we have collected as much information as possible on each of them, on their relation with the associated web servers, and on their dynamics. To do so, we have taken advantage of HARMUR, a **H**istorical **AR**chive of **M**alicious **UR**Ls also developed in the WOMBAT project.

HARMUR enables us to study the relation between client side threats and the underlying server infrastructure, and their evolution over time. Instead of developing new detection technologies (*e.g.,* based on honeyclients, or special web crawlers), HARMUR integrates multiple information sources and takes advantage of various data feeds that are dedicated to detecting Web threats. By doing so, HARMUR aims at enabling the creation of a "big picture" of the client-side threat landscape and its evolution.

In the specific context of this work, HARMUR generated the necessary contextual information on the identified rogue AV domains, and on all the other domains that were discovered to be sharing the same server as rogue AV domains thanks to DNS mapping information. In order to generate a dynamic perspective on the characteristics of the observed domains, HARMUR implements a set of analysis modules that are re-iterated on each tracked domains on a daily basis:

- Information on the security state of a domain.
  - **Norton Safeweb information.** For each domain, we have queried its security status taking advantage of the Norton Safeweb website reputation service[2]. This allowed us to retrieve information on a variety of threats known to be present on each domain, ranging from browser exploits, to malware samples, to phishing sites.

[2] `http://safeweb.norton.com`

  - **Google Safe Browsing information.** We have taken advantage of the Google Safe Browsing API[3] to detect the presence of threats within a given domain.
- Information on the domain.
  - **Registration information.** We have parsed the registration data obtained via the WHOIS protocol in order to get information on the identity of the registrant and of the provided contact email address, as well as the name of the registrar[4].
  - **DNS relations.** By means of DNS queries, we have retrieved for each domain the associated *NS* records and all the *A* records associated to all the hostnames known to belong to it. Whenever only one domain name was available and we had no information on the associated hostnames, we considered as hostnames the domain name itself and the hostname generated by prepending the standard "www" name.
- Information on the servers.
  - **Geolocation and AS information.** For each web server associated to the rogue domain through a DNS *A* record, we have collected information on its geographical location as well as its associated Autonomous System number.
  - **Server uptime and version string.** By means of HTTP HEAD packets, we have tested the responsiveness of the discovered servers and, by looking at the HTTP response headers, we have collected information on the server configuration by looking at the advertised server version string.

3.3 Limitations

Despite our efforts to maximize the threat coverage by aggregating as many information sources as possible, we are fully aware of the limitations of the dataset at our disposal. Due to the nature of our observational ability and the way the rogue AV ecosystem operates, it is impossible to know with certainty what fraction of the total rogue AV providers across the whole Internet we have been able to observe. For instance, we have noticed a predominance of servers physically located in US. This result might be skewed by the type of heuristics used for identifying rogue AV sites, that could overlook rogue AV servers that are primarily marketed to non-English languages. Moreover the identification of rogue domains is itself a potential source of bias. Our analysis is based on the identification of rogue AV domains performed by third party sources, and does not provide any guarantee in terms or precision of classification. We have

[3] `http://code.google.com/apis/safebrowsing/`

[4] The WHOIS specification [6] requires WHOIS records to be human readable, and does not specify their syntax and their semantics. As a consequence, the data stored by different registrars is often in different formats. We have built a generic parser that handles a vast number of registrars and 17 specific parser for other common registrars, but despite of this effort registration information is not available for all the domains taken into consideration.

indeed identified through manual inspection of our feeds a limited number of domains that did not seem to be actually related to the rogue AV threat type. However, the number of such misclassifications is negligible relative to the size of the dataset. Moreover, when mining the different rogue AV campaigns, any possible pollution of the dataset has been implicitly filtered out by our clustering techniques, as described later.

3.4 Multi-criteria Decision Analysis

To analyze the campaigns through which rogue AV software is distributed, we have used an *attack attribution* method that relies on a multi-criteria fusion algorithm that has proven to bring several advantages with respect to more traditional clustering methods [25]. Thanks to this method, rogue AV domains are automatically grouped together based upon common elements likely due to the same *root cause*, *i.e.,* same rogue campaign. This attribution method is based on a combination of a graph-based clustering technique with a data aggregation method inspired by multi-criteria decision analysis (MCDA). This method has been successfully used previously to analyze other types of attack phenomena [5,26,27], namely attack events found in honeypot traces.

Generally speaking, the method systematically combines different *viewpoints* such that the behavioral properties of given phenomena are appropriately modeled by the aggregation of all features.

The attribution method used in this paper consists of three components:

1. **Feature selection:** we determine which relevant features we want to include in the overall analysis, and we characterize each element of the dataset according to each extracted feature denoted by $F_k, k = 1, \ldots, n$ (*e.g.,* by creating feature vectors for each element).
2. **Graph-based clustering:** an undirected edge-weighted graph is created regarding every feature F_k, based on an appropriate distance for measuring pairwise similarities.
3. **Multi-criteria aggregation:** we combine the different graphs of features using an *aggregation function* that models the expected behavior of the phenomena under study.

The approach is mostly unsupervised, *i.e.,* it does not rely on a preliminary training phase.

Feature selection. Among the different information tracked through HARMUR, we have selected a number of features that we believed to be likely to reveal the organized operation of one specific individual or group.

- **Registrant email address (F_{Reg}).** Whenever available, the email address provided upon registration of the domain.
- **Web Server IP addresses (F_{IP}), class C ($F_{Cl.C}$), class B ($F_{CL.B}$) subnets.** To allow the identification of servers belonging to the same infrastructure, we have separately considered three features corresponding to the full IP address, its /24 and its /16 network prefix.

- **Nameserver IP address (F_{NS}).** The IP address of the authoritative nameserver(s).
- **Registered domain name (F_{Dom}).** We decided to consider as a feature the domain name itself to be able to detect common naming schemes.

In summary, by analyzing the available features, we have defined the following feature set: $\mathcal{F} = \{F_{Reg}, F_{IP}, F_{Cl.C}, F_{Cl.B}, F_{NS}, F_{Dom}\}$, which will be used by the multi-criteria method to link rogue domains to the same campaign.

Graph-based representation. In the second phase of our attack attribution method, an undirected edge-weighted similarity graph is created regarding each selected feature F_k, based on an appropriate distance for measuring pairwise similarities. A specific definition of similarity had to be defined for each of the considered features.

Since feature vectors defined for F_{IP}, $F_{Cl.C}$, $F_{Cl.B}$ and F_{NS} are simply *sets* of IP addresses (or sets of IP subnets), it is relatively easy to calculate a similarity between two sets by using the *Jaccard similarity* coefficient. This coefficient is commonly used to estimate the amount of overlap between two sets of data.

While simple equality would have been sufficient, we wanted to incorporate into F_{Reg} some additional semantics, taking into consideration the usage of specific email domains or the usage of specific keywords. For this reason, we have given maximum similarity score to identical email addresses, and non-null similarity scores to email addresses sharing same username, same email domain, or both containing popular AV keywords. For the sake of conciseness we refer the interested reader to [25] for more detailed information on this measure.

Finally, we wanted to define a notion of similarity for F_{Dom} able to catch commonalities between rogue domain names having similar patterns, or common sequences of the very same tokens. We have accomplished this goal by using the *Levenshtein distance*[5]. To normalize the Levenshtein distance to a similarity metric, we have used a commonly-used transformation [23] that maps a generic distance value to a similarity score within the interval $[0, 1]$.

Multi-criteria aggregation. As a final step of the multi-criteria analysis, we have used an *aggregation function* that defines how the criteria (*i.e.,* the site features) must be combined to group rogue domains as a function of their common elements.

An aggregation function is formally defined as a function of n arguments ($n > 1$) that maps the (n-dimensional) unit cube onto the unit interval: $f : [0, 1]^n \longrightarrow [0, 1]$. To model complex requirements, such as "most of" or "at least two" criteria to be satisfied in the overall decision function, we have used Yager's *Ordered Weighted Averaging* (OWA) [30].

Other possible aggregation functions that allow for more flexible modeling, such as the Choquet integral, may also be used and have been considered elsewhere [25].

[5] Levenshtein distance corresponds to the minimum number of operations needed to transform one string into the other (where an operation is an insertion, deletion, or substitution of a single character).

Definition 31 (OWA) [2,30] *For a given weighting vector* $\mathbf{w}$*,* $w_i \geq 0$*,* $\sum w_i = 1$*, the* OWA *aggregation function is defined by:*

$$OWA_{\mathbf{w}}(\mathbf{z}) = \sum_{i=1}^{n} w_i z_{(i)} =< \mathbf{w}, \mathbf{z}_{\searrow} > \quad (1)$$

where we use the notation $\mathbf{z}_{\searrow}$ *to represent the vector obtained from* $\mathbf{z}$ *by arranging its components in decreasing order:* $z_{(1)} \geq z_{(2)} \geq \ldots \geq z_{(n)}$*.*

In our application, the vector $\mathbf{z}$ represents the set of similarity values obtained by comparing a given pair of domains with respect to all site features F_k, as defined previously. By associating weights to the *magnitude* of the values rather than their particular inputs, OWA aggregation allows us to define a weighting vector $\mathbf{w}$ that gives lower weights to the two highest scores:

$$\mathbf{w} = [\ 0.10,\ 0.10,\ 0.20,\ 0.30,\ 0.20,\ 0.10\ ]$$

In other words, we assign more importance to features starting from the third highest position. The two highest scores will have lower weights (0.10), and thus *at least three* strong correlations will be needed to have a global score above 0.3 or 0.4, which will be used as a decision threshold to keep a link between two domains. A sensitivity analysis has been performed on this decision threshold to determine appropriate ranges of values [25]; however, due to space constraints, we do not provide further details in this paper.

4 Insights on the Rogue Security Software Threat Economy

We will now look into the details of the dataset presented in the previous section and try to infer information regarding the modus operandi of the individuals at the root cause of these businesses.

4.1 High-Level Overview

The dataset at our disposal consists of 6,500 DNS entries, collected between June and August 2009, pointing to 4,305 distinct IP addresses hosting rogue AV servers. At least 45% (2,935) of all domains were registered through only 29 Registrars.

As a first step, we have taken advantage of the DNS information at our disposal to set apart generic hosting services, hosting both rogue AV domains and benign sites, from servers specifically deployed for hosting Rogue AV content. We identified all DNS entries resolving to the same IP address, and correlated these with lists of known rogue AV- and malware-serving domains. A total of 2,677 IP addresses (web servers) host only domains that are known to serve rogue AV software. An additional 118 IPs provide services for both rogue-AV and other malware-serving domains. The remaining 1,510 IP addresses host both malicious and benign domains, and are therefore likely to be associated to hosting services unaware of the illicit use of their infrastructure.

Table 1. Top 10 server version strings

Version string	# servers
Apache	610
Microsoft-IIS/6.0	218
Apache/2.2.3 (CentOS)	135
Apache/2.2.3 (Red Hat)	123
Apache/2	100
Apache/2.2.11 (Unix) mod_ssl/2.2.11 OpenSSL/0.9.8i DAV/2 mod_auth_passthrough/2.1 mod_bwlimited/1.4 FrontPage/5.0.2.2635	69
Apache/2.0.52 (Red Hat)	49
nginx	33
Apache/2.2.11 (Unix) mod_ssl/2.2.11 OpenSSL/0.9.8e-fips-rhel5 mod_auth_passthrough/2.1 mod_bwlimited/1.4 FrontPage/5.0.2.2635	32
LiteSpeed	26
Others	1498

Table 2. Top 10 registrant email domains

Domain	# registered domains
gmail.com	1238 (30.52%)
id-private.com	574 (14.15%)
yahoo.com	533 (13.14%)
whoisprivacyprotect.com	303 (7.47%)
privacyprotect.com	125 (3.08%)
mas2009.com	101 (2.49%)
space.kz	90 (2.22%)
NameCheap.com	85 (2.10%)
domainsbyproxy.com	62 (1.53%)
hotmail.com	59 (1.45%)

Rogue AV servers localization. Mapping the 2,677 IPs hosting only rogue AV software to Autonomous System (AS) numbers, we identified a total of 509 ASes. Interestingly, but yet not surprisingly, the distribution of servers over ASes is skewed towards some specific ASes: approximately 37% (984 servers) are hosted by only 10 particularly popular ASes. As previously pointed out, the geographical distribution of these servers is heavily skewed towards US locations: approximately 53% (1,072 servers) are hosted in the USA.

Rogue AV server versions. When looking at the web server type and version for the 2,677 rogue AV web servers, in some cases we see some very specific configurations that may be indicative of the use of standardized templates or of a single entity/operator using a fixed configuration. Table 1 reports some of the most popular observed version strings. Overall, Apache (in various configurations) seems to be used in well over 40% of the rogue AV servers.

Rogue AV domain registrations. We also looked at the email addresses provided by all Registrants of rogue AV domains. The list of most popular domains, shown in Table 2, contains some of the obvious email hosting services (Gmail, Yahoo! Mail, Lycos, *etc.*). More interestingly, we see that 26% of the analyzed domains make use of anonymous domain registration services such as *domainsbyproxy.com*, *whoisprivacyprotect.com*, *id-private.com*, and *space.kz*. We also see some cases of ISPs that do not formally offer anonymous domain registration services, but are rather lax in their verification of registrant identities and email addresses. For instance, *Namecheap.com* is often associated to registrant names ranging from "Kyle" to "AA".

Rogue AV domains and browser exploits. While rogue AV software seems to be primarily trying to lure users into downloading software to stop non-existing security issues on their systems (*scareware*), we found it interesting to

evaluate the presence of other threats on the domains by correlating them with information provided by web crawlers. We determined that 814 of the rogue AV domains were serving malware of various types; 417 domains attempted to use browser exploits; 12 domains led to the installation of spyware, and 19 domains would cause the installation of a trojan. This result underlines the use, in a minority of cases, of more aggressive strategies for the monetization of clients lured into visiting the domains.

Towards the big picture. Given the size of the dataset, it is beyond the scope of this work to describe all the relationships we discovered in the course of our analysis. We have although tried to generate a "big picture" of the landscape by plotting in Figure 1 the relationships between servers hosting rogue AV content and all the domains known to be hosted on them, a total of 235,086 domains. Due to the complexity of the landscape, we have tried to simplify the visualization by omitting all IPs that were associated to less than 100 different domains. The represented domains comprise both known rogue AV domains and unrelated domains that have been discovered as being hosted on the same server thanks to

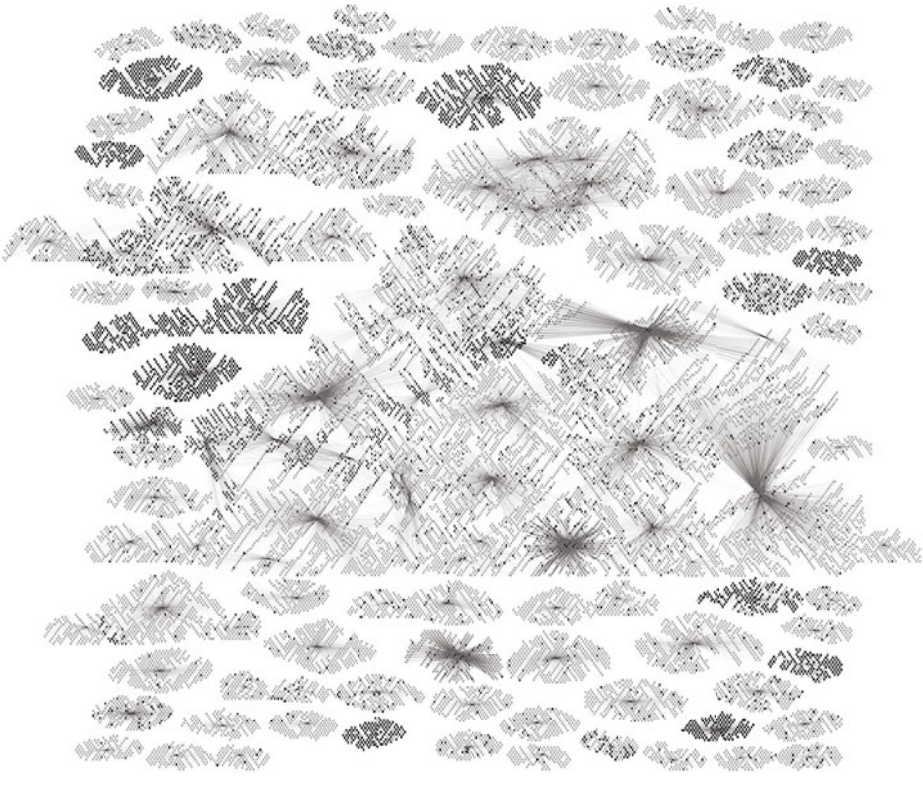


Fig. 1. Relationships between observed domains and the servers hosting them. Darker nodes represent rogue AV domains, while lighter nodes indicate benign domains.

robtex.com. We have used darker colors to differentiate rogue AV domains from the others.

The subset represented in Figure 1 consists of 174 servers that were hosting a total of 30,632 distinct domain names. In this observed domain set, 15% of the total hosted rogue security software, while 9% were observed to host other types of threats. Interestingly, most of the domain names are linked to a single web server, but some rogue AV domains were associated, over time, to several distinct IP addresses, creating some complex subgraphs such as those in the middle of Figure 1.

Figure 1 shows the complexity of the problem of the identification of malicious domains. It highlights the challenges of protecting the web clients from potentially dangerous sites through IP-based blacklisting methods. Indeed, the coexistence of both rogue and legitimate domains on the same server IP undermine the applicability of such approaches since it would be detrimental to perfectly benign sites. We will explore this issue further in Section 6.

4.2 The Campaigns

To get a better understanding of the modus operandi of the rogue AV operators, we have taken advantage of the multi-criteria decision analysis (MCDA) described in Section 3.4 to mine the dataset and identify separate campaigns likely to be generated by the action of a single individual or group.

The application of the method has led to the identification of 127 separate campaigns grouping a total of 4,549 domains. The identified campaigns have an average size of 35.8 domains, but with a significant variance in size. More specifically, 4,049 domains are associated to the 39 biggest campaigns, with the largest comprising 1,529 domains.

In the rest of this Section we will look more closely at three of these campaigns and we will show through their analysis the value of the MCDA in getting insights on the dynamics of the rogue AV threat landscape.

Large-scale campaigns. Some of the campaigns identified by our attribution method consisted of several hundreds domains. One of such examples is represented graphically in Figure 2. The graph represents the relationship between domains (clustered in big, dense groups of small rectangles), the subnets of their hosting servers (represented with larger, lighter rectangles) and the registrant email addresses (represented with large, dark rectangles). The nodes at the bottom of the graph represent instead domain registration dates.

Figure 2 groups about 750 rogue domains that have been registered in the .cn TLD (resolving to 135 IP addresses in 14 subnets), on eight specific dates over a span of eight months. However, despite the apparent link to China, the majority of the IP addresses of the hosting servers were hosted in the United States, Germany, and Belarus. In fact, no server could be identified as being located in China.

Interestingly, the same Chinese registrar (*Era of the Internet Technology*) was used for the registration of all domain names. All of the domain names are

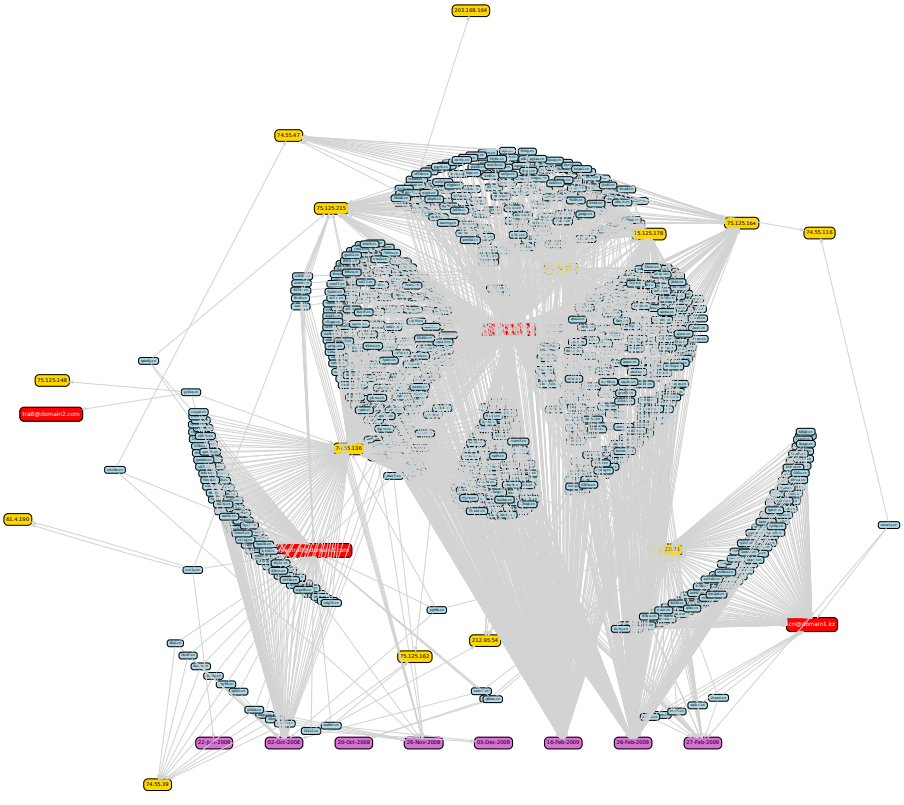


Fig. 2. Graphical representation of a long lasting, larger campaign

composed of exactly 5 alphanumeric characters, apparently chosen in a random fashion (*wxe3x.cn*, *owvmg.cn*,...), which indicates the use of automated tools to create those domains. Another noteworthy characteristic of this campaign is that the registrant responsible for 76% of the domains makes use of a WHOIS domain privacy protection service (cn@id-private.com), which we have said to be a commonly observed characteristic in certain rogue campaigns.

Finally, a manual analysis of the domains represented in Figure 2 revealed a more complex underlying ecosystem. These domains were in fact linking to a fake scan page hosted on a server belonging to a separate campaign. Such discovery underlines the existence of complex interrelations in this threat ecosystem, interrelations that would have been impossible to discover without the employment of data mining techniques able to reduce a corpus of thousands of domains to few, large campaigns carried out by specific individuals.

PC-Security and PC-Anti-Spyware campaigns. One very good example of such interrelation can be found when looking at two other distinct clusters identified by our approach. The multi-criteria decision algorithm correctly identified

them as two distinct campaigns, as they involve different features (different timings, different web servers, *etc.*). However, the analysis of both clusters reveals a common modus operandi used in both cases.

Indeed, the two clusters were composed of a relatively low number of domains (16 and 14) that were clearly referring to anti-virus or anti-spyware "products" (*e.g., pcsecurity-2009.com*, *homeav-2010.com*, *pc-antispyware2010.com*). A number of similarities could be identified in their deployment strategy:

- Both clusters use the exact same domain naming scheme, consisting of the insertion of dashes among a set of fixed words (*e.g., pc-anti-spyware-2010.com*, *pc-anti-spyware-20-10.com*, and *pc-antispyware-2010.com*).
- All of the domains in each cluster use the same registrar (OnlineNIC) and are serviced by the same two ISPs.
- The email addresses of all domain registrants are in ".ru" domains.
- The servers were on consecutive IP addresses, although the two clusters were associated to servers in completely different networks.

Perhaps even more conclusively, upon manual inspection we found that the content of each site was identical, with the exception of one differing image. All this leads us to assume that the deployment of the rogue AV domains was carried out with a good degree of automation by interacting with a single registrar. It is also worth noting that both clusters are split between two different ISPs, suggesting an attempt to provide some level of redundancy in case a cluster is taken offline by the ISP. Finally, we observed that all of the hosting web servers were located in the US. We refer the interested reader to [25] for a more detailed presentation of these two results, as well as other interesting ones.

5 Landscape Characteristics

Section 4 provided an in-depth overview of the rogue AV threat landscape, and showed our ability to identify within such landscape articulated campaigns deployed via a certain level of automation. The specificity of these characteristics to the Rogue AV landscape stays although unproved so far. This Section addresses this problem by performing a comparative analysis of the results obtained for the Rogue AV landscape with those obtained by looking at a completely different web-borne threat: drive-by downloads. We will show that the complexity of the identified campaigns is a very specific characteristic of the rogue AV landscape, and we will go further by looking into the economics of this landscape, showing that the particularly large return on investment largely justifies the complexity of the observed infrastructure.

5.1 Comparison with Drive-By Downloads

The methodology proposed so far is completely generic, and can be utilized equivalently to study the characteristics of the infrastructure underlying any web-borne threat. We have therefore decided to leverage this fact to compare

our findings for the rogue AV threat with those of a specific type of drive-by download. To do so, we have constructed a second dataset taking advantage of data generated by some internal web crawlers and used it as an additional URL feed for HARMUR. Among all the exposed threats, we have chosen to focus on all the landing sites (we use "landing site" as in [19] to denote the site that initiates a drive-by download) that exploited a very specific weakness, namely the *Internet Explorer ADODB.Stream Object File Installation Weakness* (CVE-2006-0003). We have thus repeated the very same experiment as the one performed on the rogue AV dataset, collecting information on the very same network observables and using such information to build domain features for the multi-criteria analysis technique.

While the multi-criteria approach could successfully identify 127 distinct clusters in the rogue AV dataset, 39 of which accounted for more than 60% of the domains, the clustering profile is very different when looking at the browser exploits web sites. Only 15 small clusters have been identified, accounting for only 201 domains (3.8%). This means that the vast majority of domains (96.2%) did not end up in any cluster. In other words, the very same approach that allowed us to identify large correlations within the rogue AV domains seems to be incapable of detecting any significant correlation in the browser exploit landscape. The reason for this striking difference can be found in the different modus operandi associated to these two threat classes. Our methodology aims at identifying correlations indicative of a shared ownership and management of a set of domains. In rogue security software, the infrastructure in charge of luring the victims into installing the products is maintained by the criminals themselves. This includes both the cost of registering the domains and maintaining the hardware, but also the effort of attracting the users towards it.

This does not seem to happen in the drive-by downloads threat landscape: in the vast majority of cases, the landing pages in charge of exploiting the clients are owned and maintained by uncorrelated individuals. As showed also in [19], drive-by downloads mainly operate by compromising legitimate domains that implement weak security practices. What motivates the individuals at the root of the rogue AV threat infrastructure to sustain the additional cost of directly deploying and maintaining these domains? Providing an answer to this question requires a better understanding on the costs and the revenues associated to the deployment of a rogue AV campaign.

5.2 Rogue AV Monetization

Data collection. The problem of studying the victims of online attacks has received much attention in the past few years. The crux of the problem is that attacks and victims are geographically and temporally distributed, and, as a consequence, there is generally no natural vantage point that researchers can leverage to perform their monitoring.

One approach to overcome this problem consists of performing passive measurements of the secondary effects of victims' activity. For example, in the context of spam botnets, researchers have used spam messages [31] and DNS

queries [20,22] as proxy indicators of infected machines. Active approaches are also possible. For example, in some cases, researchers have been able to infiltrate part of the attackers' infrastructure, and, thus, gain visibility of its victims directly from "the inside" [12,24].

These are interesting approaches, yet sometimes difficult to implement for practical or legal reasons. Therefore, we decided to use a novel approach to collect information "remotely from the inside".

Indeed, we observed that, in a number of cases, the servers hosting rogue AV sites were configured to collect and make publicly available statistics about their usage. These servers were Apache web servers using the `mod_status` module, which provides a page (by default, reachable at the `/server-status` location) with real-time statistics about the server status, such as the number of workers and the count of bytes exchanged. When the module is configured to generate "extended status" information, it also provides information about the requests being currently processed, in particular, the address of the client issuing a request and the URL that was requested.

We note that the server status data source has a few limitations. In particular, it does not give access to the content of the communications between clients and servers. As a result, we cannot be certain of the outcome of any access: oftentimes, we will see that the same web page (URL) is accessed, without knowing if the access was successful or not. Second, the server status page only provides the IP address of each victim. It is well known that, mostly due to NAT and DHCP effects, IP addresses are only an approximate substitute for host identifiers, and, due to the lack of visibility into the client-server traffic, we cannot use existing techniques to mitigate this problem [3,29]. Despite these limitations, the server status data allows us to gain some visibility into the access behavior of rogue AV clients.

Victim access dataset. In total, we identified 30 servers that provided status information. Of these, 6 also provided information about client requests, which is critical for our analysis. We continuously sampled the server status pages of each of these 6 servers over a period of 44 days and stored the access time, source IP address of the client, and the specific URL on the server that was accessed. The 6 servers hosted 193 active rogue AV domains, and an additional 4,031 domains, 62 of which were also rogue AV sites but did not receive any traffic. The remaining 3,969 co-located domains are a mix of malware-serving and benign sites. We then removed from our dataset requests that were not directed at rogue AV sites or that consisted only of probing or scanning attempts. After this filtering, we identified 372,096 distinct client IP addresses that interacted with the rogue AV servers during our observation period.

Localization and server usage. Clients from all around the world interacted with the rogue AV servers. The countries that were most visiting them were USA (147,729 distinct client IPs), UK (20,275), and Italy (12,413). Some rogue AV sites appear to be more popular than others, in terms of the distinct client IP addresses that were served by each. A number of sites only received a handful of

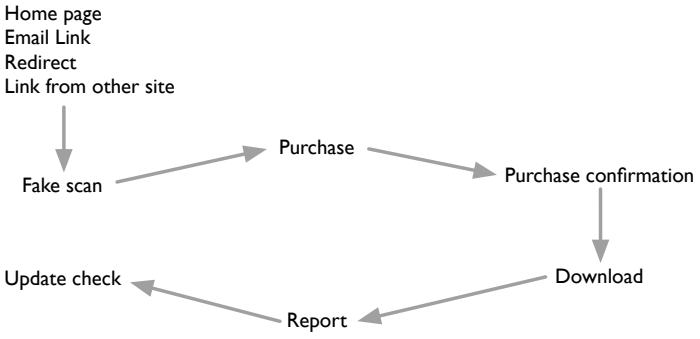


Fig. 3. Typical sequence of accesses by client

clients; in fact, 27 of the rogue AV sites were visited by only 1 client (probably an indication that these sites were no longer, or not yet, actively used). The average number of distinct client IP addresses per rogue AV site was 2,557, with a median of 560 and a standard deviation of 5,405. The 10 most popular rogue AV sites are listed in Table 3.

Access behavior. By clustering the requests issued by the clients according to the requested URL's path, we identified 6 different request types: scan, purchase, purchase confirmation, download, report, and update check. Figure 4 presents the cumulative count of distinct clients (IP addresses) that were observed issuing each type of request. (The presence of the same type of requests on different sites is probably an indication that many rogue AV sites are built on top of the same "rogue AV toolkit.")

As represented in Figure 3, these requests correspond to distinct phases with respect to the interaction of victims with rogue AV sites. A user that is somehow redirected to one of these servers is typically presented with the option to run

Table 3. Most accessed rogue AV sites

Rank	Site	Clients (#)
1	windoptimizer.com	55,889
2	inb4ch.com	23,354
3	scan6lux.com	21,963
4	gobackscan.com	19,057
5	pattle.info	14,828
6	goscansnap.com	14,590
7	goscanback.com	11,347
8	tranks.info	10,050
9	cherly.info	9,875
10	phalky.info	9,836

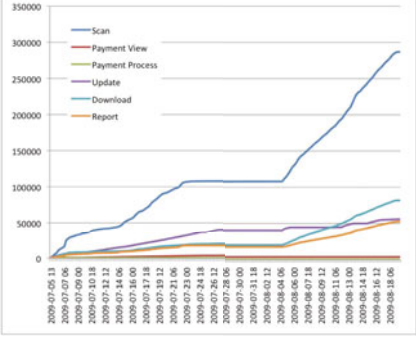


Fig. 4. Cumulative clients activity

a *scan* (typically perfunctory) of their computer. The goal of the scan is to scare users into downloading the rogue AV, by reporting that a large number of malware (viruses, Trojans, *etc.*) and other threats are present on their computers. If the scan is successful in this goal, the user will click through to a *purchase* page, or to a "free" download page. In the former case, users enter their information, including payment details (typically a credit card), and are presented with a *purchase confirmation* page. If the charge is successful, or for sites that offer a free "trial" version of rogue AV software, the user is redirected to a *download* page. Once it is successfully installed on the user's computer, the rogue AV software will periodically *check for updates* by querying a specific URL on the server. In certain cases, it will also *report* back to the server the occurrence of specific events (*e.g.,* the download of new updates). During our monitoring, each site handled only a few types of requests. More precisely, a large number of sites were devoted to handling only scan requests, while payment requests were directed at only 7 sites. We speculate that this separation of tasks simplifies the management of the campaign: even when a scan site is taken down, the processing of payments remains unaffected.

Monetization. To determine the success rate of rogue AV servers in convincing clients to download their software, we have counted the number of IPs that had performed, on the same day, a scan followed by a download. We have also counted those that did not perform a download after a scan. Doing so, we observed 25,447 successful and 306,248 unsuccessful scans, leading to the estimation of a 7.7% conversion rate from scan to download.

Similarly, our access data indicates a 1.36% conversion rate from scan to payment. Given an average price for rogue AV software of between $30 and $50, our analysis indicates that these 6 servers (which may be controlled by the same entity, or 6 distinct entities) may generate a gross income of between $111,000 and $186,000 in a period of 44 days. However, this is a best-case scenario; it is likely that at least some of the accesses to the payment URL represented failed or non-existent payments (recall that we do not have access to the actual server response). If we use a more conservative conversion rate between web server access and actual purchase of 0.26%, estimated by others in the context of email spam [13], the gross income for rogue AV site operators in the same period would range between $21,000 and $35,000. The total operational costs for these rogue AV sites would consist of the cost of hosted web servers and the cost of registering the 193 DNS domains. An informal survey of the providers hosting rogue AV sites indicates that the average monthly cost of a hosted web server is $50. Similarly, the annual domain registration costs vary between $3 and $10. Thus, the costs to the rogue AV operators would range between $1,179 and $2,530 (potentially under $400, if we pro-rate the domain registration for a 44-day period).

While the above cost estimate does not take into consideration the additional cost of advertising the maintained domains through different techniques, the costs are easily covered by the (unknown) income from other illicit activities that piggy-back on the rogue AV distribution flow (*e.g.,* keystroke loggers installed through drive-by downloads by the rogue AV servers).

Ultimately, the easiness with which rogue AV campaigns manage to successfully lure users into purchasing their products generates a return on investment that fully justifies the deployment and the management of complex infrastructures such as those studied in this work.

6 Lessons Learned and Countermeasures

This work leverages the analysis of real data to study the general characteristics and dynamics of a specific threat landscape, that of rogue security software. We identify the specificities of such threat landscape and their foundations in a particularly favorable market. Such knowledge has direct repercussions on nowadays security practices, and helps underlining weaknesses in currently employed techniques as well as potentials for new research avenues.

Users. Despite of a minor number of cases in which rogue AV domains were observed also in association to other type of threats such as drive-by downloads, the main propagation vector for this type of threat is the psychological impact on the user. The in-depth study of the reasons for the successfulness of the interaction between victims and rogue campaigns is out of the scope of this work, but our analysis clearly shows that users have an important role in the successfulness of rogue AV campaigns. As suggested in [10], the cost-benefit tradeoff associated to the offering of security services is often badly received by the users, that tend to reject the necessity of performing monetary investments to be shielded from hypothetical security threats. Rogue security software leverages this social reality to its own advantage. Increasing user awareness on the cost implicitly associated to security *may* have an impact on the relatively high conversion rates observed in this study, and *may* impact the return on investment associated to rogue AV campaigns.

Blacklisting is strained. Our study revealed two characteristics of the infrastructure used to spread rogue AV that have important consequences on the effectiveness of countermeasures against this threat, and, specifically, of blacklisting, a technique commonly used to prevent end users from accessing malicious resources.

As Figure 1 showed, the rogue AV infrastructure comprises both servers that exclusively host a very large number of rogue AV sites and servers where rogue AV sites coexist with legitimate ones. This situation is a worst case for blacklisting. In fact, IP-based blacklisting (where access to a specific web server IP is blocked) is bound to generate many false positives, thus preventing users from visiting benign sites that happen to be hosted on server IPs that also serve malicious sites. In fact, a naive IP-based blacklisting approach, listing all the servers we identified, would have incorrectly blocked access to 129,476 legitimate web sites. Conversely, domain name-based blacklisting (where access to a specific domain is blocked) is undermined by the easiness with which malicious actors can register large batches of domains. The registration of hundreds of automatically generated domain names observed in the different campaigns is likely to be an active attempt to evade such lists. For example, 77 of the rogue-specific servers

that we tracked were associated with more than twenty different domains, with a maximum of 309 domains associated to a single server.

Taking-down rogue AV campaigns. What would be a good strategy then to effectively fight rogue AV campaigns? Our analysis of the victim access dataset hinted at one possible direction: taking down payment processing sites. In fact, these appeared to be less in number than other rogue AV sites (recall that 7 payment sites supported almost 200 front-end "scanning" sites) and seemed to change less frequently. Furthermore, by disrupting the sites generating revenue, defenders are likely to significantly affect also other parts of the rogue AV operations (*e.g.*, registering new sites and paying for hosting).

DNS-based threat detection. This study has highlighted once more the important role of the DNS infrastructure in Internet threats. Rogue AV campaigns often rely on misleading DNS names to lure victims into trusting their products (*e.g.*, *pcsecurity-2009.com*). Also, we have seen how such campaigns often lead to the automated deployment of large numbers of domains pointing to a few servers and following well-defined patterns in their naming schema. For all these reasons, as already noted in [20] for other type of threats, DNS seems to be a promising point of view for the detection of such anomalies.

7 Conclusion

We presented a longitudinal analysis on the infrastructure and the dynamics associated with an increasingly popular threat, that of rogue security software.

The contributions of this paper are threefold. Firstly, we provide the first *quantitative* high-level analysis of the rogue AV threat landscape and the underpinning infrastructure. We detail the relationships between rogue AV domains and the web servers hosting them, and we delve into their characteristics to extract high-level information on the structure of these threats. Secondly, we apply a threat attribution methodology to 6,500 domains under observation and we automatically extract information on large-scale campaigns likely to be associated to the operation of a single individual or group, likely through the help of automated tools. Finally, we provide insights on the economy of the rogue AV threat landscape by leveraging information on the interaction of victim clients with several rogue AV servers over a period of 44 days. We show how the rogue AV distributors are able to generate considerable revenues through their activities, which fully justifies their investment in the deployment of the distribution infrastructures.

While this paper targets specifically the rogue antivirus threat, we believe that the methodologies and the lessons learnt from our work can be of value to the study of other threats (*e.g.,* phishing and other scams). More specifically, we show how the combination of clustering and data aggregation methods can be leveraged to profile different threat landscapes and, by comparison, offer a valuable tool to the study of threat economies.

References

1. Microsoft Security Intelligence Report, volume 7. Technical report, Microsoft (2009)
2. Beliakov, G., Pradera, A., Calvo, T.: Aggregation Functions: A Guide for Practitioners. Springer, Berlin (2007)
3. Bellovin, S.: A Technique for Counting NATted Hosts. In: Proc. of the Internet Measurement Conference (2002)
4. Correll, S.P., Corrons, L.: The business of rogueware. Technical Report, PandaLabs (July 2009)
5. Dacier, M., Pham, V., Thonnard, O.: The WOMBAT Attack Attribution method: some results. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 19–37. Springer, Heidelberg (2009)
6. Daigle, L.: WHOIS protocol specification. RFC 3912 (September 2004)
7. Fossi, M., Johnson, E., Turner, D., Mack, T., Blackbird, J., McKinney, D., Low, M.K., Adams, T., Laucht, M.P., Gough, J.: Symantec Report on the Underground Economy. Technical Report, Symantec (2008)
8. Fossi, M., Turner, D., Johnson, E., Mack, T., Adams, T., Blackbird, J., Low, M.K., McKinney, D., Dacier, M., Keromytis, A., Leita, C., Cova, M., Overton, J., Thonnard, O.: Symantec report on rogue security software. Whitepaper, Symantec (October 2009)
9. Franklin, J., Paxson, V., Perrig, A., Savage, S.: An Inquiry into the Nature and Causes of the Wealth of Internet Miscreants. In: Proc. of the ACM Conference on Computer and Communications Security (2007)
10. Herley, C.: So long, and no thanks for the externalities: the rational rejection of security advice by users. In: Proc. of the 2009 New Security Paradigms Workshop (NSPW), pp. 133–144. ACM, New York (2009)
11. Holz, T., Engelberth, M., Freiling, F.: Learning More about the Underground Economy: A Case-Study of Keyloggers and Dropzones. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 1–18. Springer, Heidelberg (2009)
12. Holz, T., Steiner, M., Dahl, F., Biersack, E., Freiling, F.: Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In: Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (2008)
13. Kanich, C., Kreibich, C., Levchenko, K., Enright, B., Voelker, G., Paxson, V., Savage, S.: Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In: Proc. of the ACM Conference on Computer and Communications Security (2008)
14. Krebs, B.: Massive Profits Fueling Rogue Antivirus Market. In: Washington Post (2009)
15. McGrath, K., Gupta, M.: Behind Phishing: An Examination of Phisher Modi Operandi. In: Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (2008)
16. Moore, T., Clayton, R.: Examining the Impact of Website Take-down on Phishing. In: Proc. of the APWG eCrime Researchers Summit (2007)
17. Moshchuk, A., Bragin, T., Gribble, S.D., Levy, H.M.: A Crawler-based Study of Spyware on the Web. In: Network and Distributed System Security Symposium, pp. 17–33 (2006)
18. O'Dea, H.: The Modern Rogue — Malware With a Face. In: Proc. of the Virus Bulletin Conference (2009)

19. Provos, N., Mavrommatis, P., Rajab, M., Monrose, F.: All Your iFRAMEs Point to Us. In: Proc. of the USENIX Security Symposium (2008)
20. Rajab, M., Zarfoss, J., Monrose, F., Terzis, A.: A Multifaceted Approach to Understanding the Botnet Phenomenon. In: Proc. of the Internet Measurement Conference (2006)
21. Rajab, M.A., Ballard, L., Mavrommatis, P., Provos, N., Zhao, X.: The Nocebo Effect on the Web: An Analysis of Fake Anti-Virus Distribution. In: Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (2010)
22. Ramachandran, A., Feamster, N., Dagon, D.: Revealing Botnet Membership Using DNSBL Counter-Intelligence. In: Proc. of the Workshop on Steps to Reducing Unwanted Traffic on the Internet, SRUTI (2006)
23. Shepard, R.N.: Multidimensional scaling, tree fitting, and clustering. Science 210, 390–398 (1980)
24. Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., Vigna, G.: Your Botnet is My Botnet: Analysis of a Botnet Takeover. In: Proc. of the ACM Conference on Computer and Communications Security (2009)
25. Thonnard, O.: A multi-criteria clustering approach to support attack attribution in cyberspace. PhD thesis, École Doctorale d'Informatique, Télécommunications et Électronique de Paris (March 2010)
26. Thonnard, O., Mees, W., Dacier, M.: Addressing the attack attribution problem using knowledge discovery and multi-criteria fuzzy decision-making. In: KDD 2009, 15th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Workshop on CyberSecurity and Intelligence Informatics, Paris, France, June 28-July 1 (December 2009)
27. Thonnard, O., Mees, W., Dacier, M.: Behavioral Analysis of Zombie Armies. In: Czossek, C., Geers, K. (eds.) The Virtual Battlefield: Perspectives on Cyber Warfare. Cryptology and Information Security Series, vol. 3, pp. 191–210. IOS Press, Amsterdam (2009)
28. Wang, Y.-M., Beck, D., Jiang, X., Roussev, R.: Automated Web Patrol with Strider HoneyMonkeys. Technical Report MSR-TR-2005-72, Microsoft Research (2005)
29. Xie, Y., Yu, F., Achan, K., Gillum, E., Goldszmidt, M., Wobber, T.: How Dynamic are IP Addresses? In: Proc. of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM (2007)
30. Yager, R.: On ordered weighted averaging aggregation operators in multicriteria decision-making. IEEE Trans. Syst. Man Cybern. 18(1), 183–190 (1988)
31. Zhuang, L., Dunagan, J., Simon, D., Wang, H., Osipkov, I., Hulten, G., Tygar, J.: Characterizing Botnets from Email Spam Records. In: Proc. of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (2008)

Fast-Flux Bot Detection in Real Time

Ching-Hsiang Hsu[1], Chun-Ying Huang[2], and Kuan-Ta Chen[1]

[1] Institute of Information Science, Academia Sinica
[2] Department of Computer Science and Engineering,
National Taiwan Ocean University

Abstract. The fast-flux service network architecture has been widely adopted by bot herders to increase the productivity and extend the lifespan of botnets' domain names. A fast-flux botnet is unique in that each of its domain names is normally mapped to different sets of IP addresses over time and legitimate users' requests are handled by machines other than those contacted by users directly. Most existing methods for detecting fast-flux botnets rely on the former property. This approach is effective, but it requires a certain period of time, maybe a few days, before a conclusion can be drawn.

In this paper, we propose a novel way to detect whether a web service is hosted by a fast-flux botnet *in real time*. The scheme is unique because it relies on certain intrinsic and invariant characteristics of fast-flux botnets, namely, 1) the request delegation model, 2) bots are not dedicated to malicious services, and 3) the hardware used by bots is normally inferior to that of dedicated servers. Our empirical evaluation results show that, using a passive measurement approach, the proposed scheme can detect fast-flux bots in a few seconds with more than 96% accuracy, while the false positive/negative rates are both lower than 5%.

Keywords: Botnet, Request delegation, Document fetch delay, Processing delay, Internet measurement, Supervised classification.

1 Introduction

A botnet is a collection of compromised Internet hosts (a.k.a. bots), that have been installed with remote control software developed by malicious users. Such software usually starts automatically when a parasite host boots. As a result, malicious users (a.k.a. bot herders), can coordinate large-scale Internet activities by controlling the bots (the victims). Bot herders always attempt to compromise as many hosts as possible. According to the report of the FBI's "Operation Bot Roast" project [7], more than one million victim IP addresses had been identified on the Internet by the end of 2007, and the number continues to increase. Botnets allow bot herders to engage in various malicious activities, such as launching distributed denial of service (DDoS) attacks, sending spam mails [24], hosting phishing sites [13], and making fraudulent clicks [5]. Statistics show that botnets yield great economic benefits for bot herders [16, 15]; for example, Gartner [8] estimated that the economic loss caused by phishing attacks alone is as much as US$3 billion per year.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 464–483, 2010.

To help legitimate users avoid malicious services (mostly in the form of websites) hosted on a bot, researchers and practitioners have investigated ways to determine whether a host is part of a botnet [10, 11, 9, 20]. If a bot is detected, the host owner can remove the remotely controlled software by using malicious software removal tools, or the network ISP can disconnect the bot if the host owner does not take appropriate action. Obviously, bot herders take countermeasures to keep their botnets alive and productive. Particularly, the Fast-Flux Service Network (FFSN) architecture has been used to *increase the productivity* and *extend the lifetime of domain names* linked to the bots.

Usually, a bot herder applies for a domain name for each of his bots and distributes the domain names (normally in the form of URLs) via various channels, such as spam mails or web blogs. However, if a machine is in down time, the bot cannot be controlled and the URL will be temporarily unavailable. Moreover, control of the bot may be lost due to removal of the malicious software. In this case, the bot herder will not gain any more benefits from the domain name unless it is re-mapped to another IP address (of another bot).

An FFSN-based botnet (called a *fast-flux botnet* for short), solves the abovementioned problems because of two architectural innovations: 1) the mapping between domain names and IP addresses, and 2) the way legitimate users' requests are processed.

- First, in a fast-flux botnet, *a domain name is mapped to a number of IP addresses* (possibly hundreds, or even thousands) rather than a single IP address. As a result, if the mapping is handled properly, i.e., a domain name is always resolved to a controllable and live bot, the productivity (in terms of the access rate of malicious services) will be higher than that of a traditional botnet. In addition, if it is known that a bot has been detected, the domain name's link to the bot can be terminated immediately so that their relationship cannot be discovered.
- Second, *legitimate users' requests are indirectly handled by other machines called motherships, rather than the bots the users contact.* In other words, when a legitimate user accesses a service provided by a fast-flux botnet via a URL, the bot that the URL connects to and receives requests from does not handle the requests itself. Instead, it serves as a proxy by delegating the requests to a mothership, and then forwarding the mothership's responses to the user. By so doing, bot herders can update a malicious service (and the content it offers) anytime because they have more control over the mothership and the number of mothership nodes is relatively small compared to that of bots. In addition, since malicious services do not reside on bots, it is easier for bot herders to reduce the footprint of the malicious software so that it is less likely to be detected by anti-malware solutions.

To obscure the link between a domain name and the IP addresses of available bots, fast-flux botnets often employ a strategy that resolves a domain name to different sets of IP addresses over time. For example, we observed that the malicious service `f07b42b93.com`, which hosts a phishing webpage that deceives

— Returned DNS records at time t —

```
;; ANSWER SECTION:
f07b42b93.com.    300    IN A    68.45.212.84
f07b42b93.com.    300    IN A    68.174.233.245
f07b42b93.com.    300    IN A    87.89.53.176
f07b42b93.com.    300    IN A    99.35.9.172
f07b42b93.com.    300    IN A    116.206.183.29
f07b42b93.com.    300    IN A    174.57.27.8
f07b42b93.com.    300    IN A    200.49.146.20
f07b42b93.com.    300    IN A    204.198.77.248
f07b42b93.com.    300    IN A    207.112.105.241
f07b42b93.com.    300    IN A    209.42.186.67
```

— Returned DNS records at time t+300 second —

```
;; ANSWER SECTION:
f07b42b93.com.    300    IN A    64.188.129.99
f07b42b93.com.    300    IN A    69.76.238.227
f07b42b93.com.    300    IN A    69.225.51.55
f07b42b93.com.    300    IN A    76.10.12.224
f07b42b93.com.    300    IN A    76.106.49.207
f07b42b93.com.    300    IN A    76.127.120.38
f07b42b93.com.    300    IN A    76.193.216.140
f07b42b93.com.    300    IN A    99.35.9.172
f07b42b93.com.    300    IN A    200.49.146.20
f07b42b93.com.    300    IN A    204.198.77.248
```

Fig. 1. An example of how a fast-flux botnet rapidly changes the mapping of IP addresses to its domain names. These two consecutive DNS lookups are 300 seconds apart.

users by getting them to reveal their iPhone serial numbers, adopts this strategy. As shown in Fig. 1, during a DNS query at time t, the domain's DNS server replies with 10 `A` records, any of which will lead users to the phishing webpage. The short time-to-live (TTL) value, i.e., 300 seconds, indicates that the records will expire after 300 seconds, so a new DNS query will then be required. At t+300 seconds, we re-issued the same query and obtained another set of IP addresses. In total, there are 19 IP addresses with one duplication in the two sets, which indicates that the bot herder currently owns a minimum of 19 bots. The duplication could occur because the DNS server returns IP addresses randomly, or the bot herder does not have enough bots and cannot provide any more unseen IP addresses. A single fast-flux botnet domain name may be resolved to a huge number of IP addresses. For example, we observed a total of $5,532$ IP addresses by resolving the domain name `nlp-kniga.ru` between October 2009 and March 2010. The larger the IP address pool, the higher will be the "productivity" of the botnet. As a result, the link between any two bots that serve the same bot herders will be less clear, which is exactly what the bot herders desire.

A number of approaches have been proposed to detect fast-flux botnets. By definition, a fast-flux botnet domain name will be resolved to different IP addresses over time because 1) bots may not be alive all the time; and 2) bot herders want the links between the bots to be less obvious. Therefore, most studies rely on the number of IP addresses of a domain name by actively querying a certain domain name [3, 12] or passively monitoring DNS query activities for a specific period [25] (normally a few days). This approach is straightforward and robust; however, the time required to detect bots is simply too long. A bot herder may only require a few minutes to set up a new domain name and a malicious service to deceive legitimate users; therefore, we cannot spend a few days trying to determine whether a certain domain hosts malicious services. *In order to fully protect legitimate users so that they do not access malicious services unknowingly, we need a scheme that can detect whether a service is hosted by a fast-flux botnet in real time.*

In this paper, we propose such a scheme. The key features of the scheme are as follows:

1. The scheme can work in either a *passive* or an *active* mode. In the passive mode, it works when users are browsing websites; while in the active mode, it can also issue additional HTTP requests and thereby derive more accurate decisions. Irrespective of the mode used, the scheme can determine whether a website is hosted by a fast-flux bot within a few seconds with a high degree of accuracy.
2. The scheme relies on certain intrinsic and invariant characteristics of fast-flux botnets, namely, i) the request delegation model; ii) bots have "owners," so they may not be dedicated to malicious services; and iii) the network links of bots are not normally comparable to those of dedicated servers. Among the characteristics, the first one exists by definition; while the other two, fortunately, cannot be manipulated by bot herders. Consequently, bot herders cannot implement countermeasures against the scheme.
3. The scheme does not assume that a fast-flux botnet owns a large number of bots (IP addresses). Thus, even if a botnet only owns a few bots, as long as it adopts the "request delegation" architecture, our scheme can detect it without any performance degradation.

The remainder of this paper is organized as follows. In Section 2, we discuss existing solutions for detecting fast-flux botnets. The intrinsic properties of fast-flux botnets are analyzed in Section 3. The proposed solution is introduced in Section 4. Section 5 evaluates the proposed solution. Section 6 considers practical issues related to the proposed solution. Section 7 contains some concluding remarks.

2 Related Work

To the best of our knowledge, the Honeynet project [22] was the first research to study the abuse of fast-flux botnets. The authors explained the hidden operations

of botnets by giving examples of both single and double fast-flux mechanisms. Single fast-flux mechanisms change the `A` records of domains rapidly, while double fast-flux techniques change both the `A` records and the `NS` records of a domain frequently.

Holz et.al. [12] monitored domain name service (DNS) activities over a seven-week period and proposed a fast-flux botnet domain name detection scheme based on the fluxy-score. The score is computed by counting the number of unique `A` records in all DNS lookups, the number of `NS` records in a single DNS lookup, and the number of unique autonomous system numbers (ASNs) for all DNS `A` records. A number of detection schemes [17, 18, 25, 14] detect fast-flux botnet domain names by monitoring how frequently a domain name changes its corresponding IP addresses. However, these solutions often have to observe DNS activities for a long time (months). Although the observation period can be reduced by using both active and passive monitoring techniques [3], the approach still needs several minutes along with the help of a data center to determine whether a domain name is controlled by a botnet.

The proposed fast-flux botnet detection scheme is fundamentally different from all previous approaches. Since DNS-based detectors often require a long time to identify fast-flux botnets, the proposed solution does not monitor DNS activities. Instead, it relies on several basic properties that are measured at the network level with a short period of time. As a result, it can detect fast-flux botnets accurately and efficiently.

3 Intrinsic Characteristics of Fast-Flux Bots

In this section, we consider the intrinsic characteristics of fast-flux bots, which serve as the basis of the proposed detection method described in Section 4. Since these characteristics are intrinsic and invariant, they are common to fast-flux bots. Therefore, bot herders cannot manipulate them in order to evade detection by the proposed scheme.

3.1 Request Delegation

As mentioned in Section 1, a fast-flux bot does not process users' requests itself. Instead, it acts as a proxy by delegating requests to the mothership, and then forwards responses from the mothership to the users. The purpose of this design is twofold: 1) to protect the mothership from being exposed or detected; and 2) to avoid having to replicate malicious services and content to every bot, which would increase the risk of being detected and also slow down the collection of fraudulent information (e.g., obtaining users' confidential data via phishing). The request delegation design is illustrated in Fig. 2. When a client sends a request to a fast-flux bot, the request is redirected to a mothership node, as shown in the figure. The node processes the request (mostly by reading a static webpage from a hard disk), and sends the response to the bot. The bot, as a proxy, forwards the response to the requester as if it had handled the request itself.

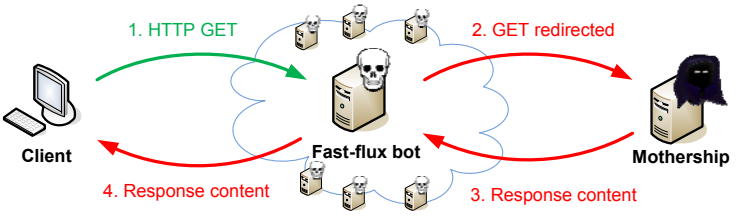


Fig. 2. An example of how a fast-flux botnet delivers malicious content secretly to a client

Because of this design, a client may perceive a slightly longer delay between issuing a request and receiving the response when the "service provider"[1] is a fast-flux bot. The increase in the response time is roughly the same as the message forwarding delay between the bot and the mothership. As long as the request delegation model is employed, technically, the increase in response time cannot be avoided.

3.2 Consumer-Level Hardware

Bot herders expand their collection of bots by compromising as many computers as possible. Most botnets are comprised of residential PCs [23]. One reason is that such PCs are not well-maintained normally; e.g., the anti-virus software may be out-of-date and/or the operating system and applications may not be patched. Residential PCs are normally equipped with consumer-level hardware and usually connect to the Internet via relatively low-speed network links, e.g., ADSL and a cable modem. As a result, compared to dedicated web servers, like those of Google and Yahoo, most bots have relatively low computation power and network bandwidth to access the Internet, which may cause the following phenomena:

- Because of a bot's relatively low computation power, the message forwarding operation at a bot may experience significant delays if any foreground application is running at the same time (see the next subsection).
- Because of a bot's relatively low network bandwidth, and the fact that residential network links are normally shared by a number of users (e.g., users in the same building), it is likely that significant network queuing will occur. This will induce variable queuing time and make a request's response time more fluctuating.

Obviously, bot herders cannot alter the level of a bot's equipment for network bandwidth access. For this reason, we consider such characteristics intrinsic and the phenomena are unalterable by external parties; in other words, longer message forwarding delays and more variable network delays should be widely observable in fast-flux botnets.

[1] We use the term "service provider" because, although a fast-flux bot is the service provider from the end-user's viewpoint, the actual service is provided by the mothership behind the bot.

3.3 Uncontrollable Foreground Applications

Ideally, bot herders should be able to control bots via remote control software; however, bots are not controlled *exclusively* by bot herders: They are personal computers that may be used by the owners at the same time. For example, a bot may be serving phishing webpages for bot herders at exactly the same time that the PC owner is playing an online game or watching a movie. This possibility indicates that foreground applications run by bot owners and background malicious processes run by bot herders may compete for computing resources, such as the CPU, memory, disk space, and network bandwidth. In other words, if the workloads of bot owners and bot herders compete for resources, the performance of both applications may suffer.

This characteristic implies that the delay incurred by the message forwarding operation at a bot, i.e., the time taken to forward a user's request to the mothership and the time taken to forward the mothership's response to the user, may vary according to the instantaneous foreground workload on the bot. This effect would be especially significant if a bot's computation power is low (due to consumer-level hardware). In this case, any foreground workload would slow the above message forwarding operation, so a high level of variability in message forwarding delays will be observed.

Bot herders cannot avoid this situation because malicious software would be easily detected if it affects the performance of bot owners' foreground applications. More specifically, if a bot herder's malicious software requests a high priority for computation, bot owners may notice that the performance of their foreground applications deteriorates and run a scan, which would detect and remove the malicious software.

3.4 Summary

In Table 1, we list the characteristics that are intrinsic to fast-flux bots, and also compare fast-flux bots with dedicated severs and traditional bots (i.e., bots that malicious services are running on, but they do not delegate users' requests). It is clear that dedicated servers do not have any of the characteristics of fast-flux

Table 1. Comparison of the intrinsic characteristics of bots (dedicated servers, traditional bots and fast-flux bots)

	Dedicated servers	Traditional bots	Fast-flux bots	Consequence
Requests delegated	×	×	✔	Long delays in fetching documents
Comsumer-level hardware	×	✔	✔	Low bandwidth & variable network delays
Uncontrollable foreground tasks	×	✔	✔	Long processing delays

bots. Traditional bots, on the other hand, are similar to fast-flux bots, except that they do not delegate requests.

The effects of these intrinsic characteristics are also summarized in Table 1. Because of these properties, we expect to see long delays in fetching documents (called document fetch delays hereafter), variable network queuing delays, and long processing delays when users make requests to a malicious service hosted by a fast-flux bot. Measuring the three types of delay form the basis of our fast-flux bot detection scheme, which we discuss in detail in the next section.

4 The Proposed Solution

In this section, we introduce the proposed solution for detecting fast-flux bots. Our scheme assumes that bot herders exploit the bots to execute web-based malicious services, e.g., phishing pages or other types of fraudulent webpages. Specifically, the malicious software on the bots includes a HTTP server that listens to TCP ports 80/443 and accepts HTTP/HTTPS requests. Before describing the proposed scheme, we explain the rationale behind our design:

- *Realtimeness.* We expect the scheme to be able to detect fast-flux bots in real time, e.g., within a few seconds, so that we can prevent legitimate users from proceeding with malicious services in time.
- *Robustness.* We expect that the scheme will not be dependent on the signatures of certain botnet implementations. The scheme must be signature-independent in order to cope with updates from existing botnets as well as new, unknown botnet implementations without degrading the detection performance.
- *Lightweight.* We expect the scheme to be as lightweight as possible so that it can be deployed on any type of device without using too many computing resources.

Given the above guidelines, we propose *a real-time, signature-less, and lightweight detection scheme* for fast-flux bots based on their intrinsic characteristics (cf. Section 3). Under the scheme, if a client tries to download webpages from a web server suspected of being a fast-flux bot, the scheme will monitor the packet exchanges between the client and the server and issue additional HTTP requests if necessary. The decision about whether the server is part of a fast-flux botnet is based on measurements of the packet transmission and receipt times observed at the client. We call the web server that the client sends HTTP requests to a "suspect server" or simply a "server." However, the machine may only be a proxy, so it does not handle HTTP requests itself (e.g., in the case of fast-flux bots).

Next, we define the three delay metrics used to determine whether a suspect server is a fast-flux bot.

1. **Network delay (ND)**: The time required to transmit packets back and forth over the Internet between the client and the server.

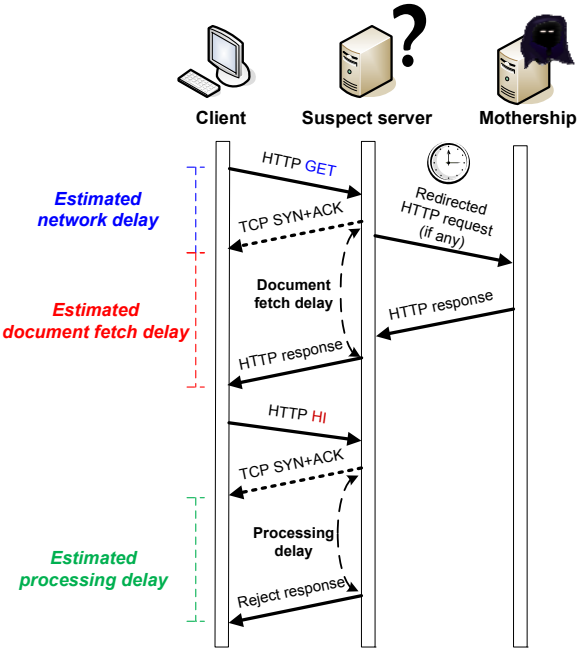


Fig. 3. The measurement techniques used to estimate network delays, processing delays, and document fetch delays based on HTTP requests

2. **Processing delay (PD)**: The time required for the server to process a dummy HTTP request that does not incur any additional computation and I/O operations.
3. **Document fetch delay (DFD)**: The time required for the server to fetch a webpage (either from a hard disk or from a back-end mothership).

Network delays occur at the network-level, while the processing delays occur at the host-level (i.e., at the suspect server). Document fetch delays are more complicated in that they may occur at the host-level only (at the suspect server) if the request delegation model is not employed, or they may arise if the server delegates received requests to a mothership via the Internet. In the latter case, DFDs involve host-level delays (at the suspect server and the mothership node) and network-level delays (between the suspect server and the mothership node). The measurement techniques used to estimate the three types of delay are shown in Fig. 3. We discuss the techniques in detail in the following sub-sections.

4.1 Network Delay Measurement

Network delay (ND) is defined as the difference between the time a client sends out the first TCP SYN packet to the suspect server and the time the client receives the corresponding TCP SYN+ACK packet from the server. By using

this estimate, a TCP connection only yields one network delay sample. To collect more samples, when appropriate, our scheme temporarily disables the persistent connection option in HTTP 1.1, which ensures a separate TCP connection for each HTTP request; thus, the number of ND samples will be the same as the number of HTTP requests.

4.2 Processing Delay Measurement

Measuring processing delays (PD) at the suspect server is not straightforward because HTTP does not support such operations naturally. We need a HTTP command that will respond to the client without contacting the back-end mothership (if any), irrespective of whether the suspect server is a fast-flux bot. For this purpose, we attempted to make the following requests:

1. Valid HTTP requests with methods other than `GET`, e.g., `OPTIONS` and `HEADER` methods.
2. HTTP requests with an invalid version number.
3. HTTP requests with incomplete headers.
4. HTTP requests with an undefined method, e.g., a nonsense `HI` method.

Our experiments showed that most fast-flux bots still contacted their mothership in the first three scenarios. On the other hand, most of them rejected HTTP requests with undefined methods directly by sending back a HTTP response, usually with the status code 400 (Bad Request) or 405 (Method Not Allowed).

Consequently, we estimate the processing delay at the server by subtracting the network round-trip time from the application-level message round-trip time. Specifically, assuming AD is the difference between the time a client sends out a HTTP request with an undefined method and the time the client receives the corresponding HTTP response (code 400 or 405), then a PD sample is estimated by subtracting ND (the network delay) from AD.

4.3 Document Fetch Delay Measurement

We define the document fetch delay (DFD) as the time required for the suspect server to "fetch" a webpage. Since the fetch operation occurs at the server side, we cannot know exactly what happens on the remote server. Thus, we employ the following simple estimator. Assuming RD is the difference between the time a client sends out a successful HTTP GET request and the time the client receives the corresponding HTTP response (code 200), then a DFD sample is estimated by subtracting ND (the network delay) from RD. Figure 4 shows the distribution of DFD, PD and their respective standard deviations measured for benign servers, traditional bots, and fast-flux bots.

4.4 Decision Algorithm

In this sub-section, we explain how we utilize the three delay metrics in our decision algorithm.

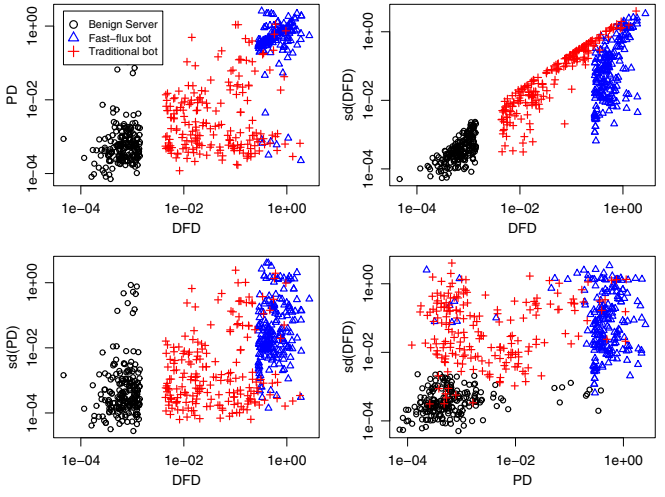


Fig. 4. Scatter plots of processing delays, document fetch delays, and their respective standard deviations. Both the x- and y-axis are in log scale.

- The objective of measuring network delays is to capture the level of network congestion between a client and the suspect server. As per Section 3.2, the ND and sd(ND), where sd(·) denotes the standard deviation, tend to be (relatively) large if the suspect server is a fast-flux bot rather than a benign, dedicated web server.
- The processing delay helps us determine the server's workload and the required computation power. If there are other workloads on the server, the estimated processing delays would be high and fluctuate over time. Thus, as per Section 3.2 and Section 3.3, the PD and sd(PD) tend to be large if the suspect server is a fast-flux bot.
- The document fetch delay indicates how much time the server takes to fetch a webpage. Because of the request delegation model (Section 3.1), DFD and sd(DFD) tend to be large if the suspect server is a fast-flux bot.

For a suspect server, we collect six feature vectors (ND, PD, DFD, and their respective standard deviations), each of which contains n elements assuming n HTTP GET requests are issued. For the PD samples, another n HTTP requests with an undefined method must also to be issued.

To determine whether a suspect server is a fast-flux bot, which is a binary classification problem, we employ a supervised classification framework and use linear SVM [4] as our classifier. A data set containing the delay measurement results for both benign web servers and web servers hosted on fast-flux bots is used to train the classifier. When a client wishes to browse pages on an unknown website, our scheme collects the delay measurements and applies the classifier to determine whether the suspect server is part of a fast-flux botnet.

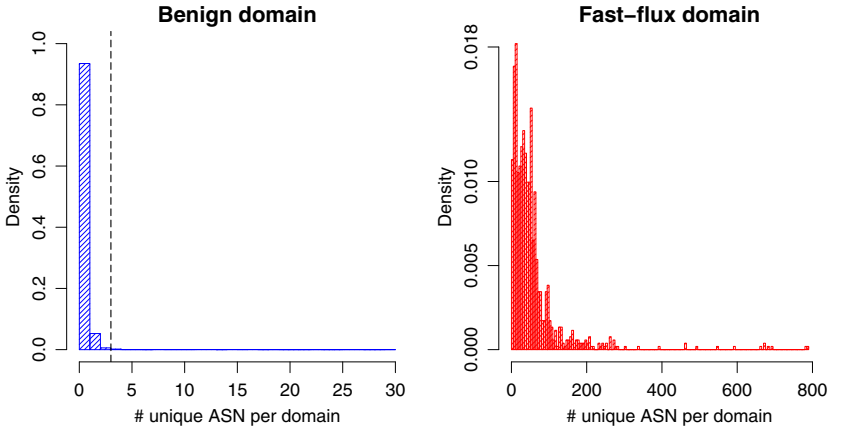


Fig. 5. The distribution of unique autonomous system numbers (ASNs) per domain name in the dataset of benign servers and fast-flux bots

5 Methodology Evaluation

In this section, we evaluate the performance of the proposed fast-flux bot detection scheme. First, we describe the data set and examine whether the derived features differ significantly according to the type of suspect server. Then, we discuss the detection performance of the scheme and consider a passive use of the scheme.

5.1 Data Description

To evaluate the performance of the proposed scheme in real-life scenarios, we need a set of URLs that legitimate users can browse. Our dataset contains the following three categories of URLs, which point to different kinds of servers:

- *Benign servers*: The top 500 websites listed in the Alexa directory [1].
- *Traditional bots*: URLs that appear in the PhishTank database [19] with suspicious fast-flux domains removed (see below).
- *Fast-flux bots*: URLs that appear in the ATLAS Global Fast Flux database [2] and the FastFlux Tracker at `abuse.ch` [6].

Between January and April 2010, we used `wget` to retrieve the URLs in our dataset at hourly intervals. During the web page retrieval process, we ran `tcpdump` to monitor all the network packets sent from and received by the client. After retrieving each web page, we sent out a HTTP request with the undefined method "HI" to measure the processing delays that occurred at the suspect server, as described in Section 4.2.

We found that some URLs in the PhishTank database actually point to fast-flux bots, and some URLs listed as pointers to fast-flux bots may actually point

Table 2. The trace used to evaluate the detection performance of the proposed scheme

Host type	#domain	#IP address	#session	#connection
Benign servers	500	3,121	60,936	565,466
Traditional bots	16,317	9,116	79,694	943,752
Fast-flux bots	397	3,513	213,562	726,762

to traditional bots. Therefore, after collecting the data, we performed a post hoc check based on the number of distinct autonomous system numbers (ASNs). Figure 5 shows the distributions of distinct ASNs of benign domain names and fast-flux domain names over the trace period. Nearly all the benign domain names were associated with three or fewer ASNs, while most fast-flux domain names were associated with many more ASNs over the three-month period. Based on this observation, we set 3 ASNs as the threshold to determine whether or not a domain name was associated with a fast-flux botnet. Thus, if a domain name was reported as a non-fast-flux bot, but it was associated with four or more ASNs (or vice versa), we regarded the domain name as questionable. We simply removed such domain names from our trace to ensure its clarity and correctness. In addition, if a URL was unavailable due to domain name resolution failures, packet unreachable errors, HTTP service shutdown, or removal of corresponding web services for 10 successive attempts, we removed it from the dataset.

The three-month trace is summarized in Table 2, where a connection refers to a TCP connection, a session refers to a complete web page transfer (including the HTML page and its accessory files, such as images and CSS files). As we turned off the HTTP 1.1 persistent connection option in order to acquire more samples for the delay metrics (cf. Section 4), the number of connections is much higher than that of sessions because a web page often contains several accessory files

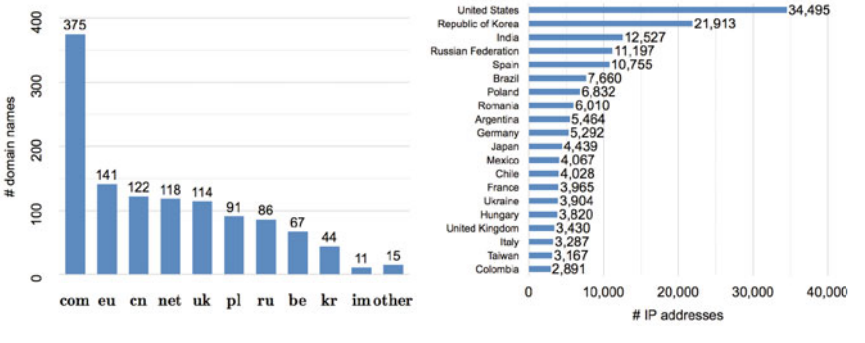


Fig. 6. (a) The top 10 top-level domains and (b) the top 20 countries associated with the fast-flux domain names in our dataset.

(maybe even dozens). Figure 6 shows the top 10 (out of 19) top-level domains and the top 20 (out of 127) countries associated with the observed fast-flux bots.

5.2 A Closer Look at the Derived Features

We now examine whether the empirical delay measurements derived during web browsing can be used to distinguish between fast-flux bots and benign servers. First, we investigate whether, as expected, consumer-level hosts incur higher and more variable processing delays and more variable network delays (cf. Section 3.4). To do this, we use a common technique that infers whether a host is associated with dial-up links, dynamically configured IP addresses, or other low-end Internet connections based on the domain name of reverse DNS lookups [21]. For example, if a host's domain name contains strings like "dial-up," "adsl," and "cable-modem," we assume that the host is for residential use and connects to the Internet via relatively slow links. Figure 7 shows the distributions of the six features for normal and consumer-level hosts. The plots fit our expectation that consumer-level hosts of fast-flux botnet incur more variable network delays, longer processing delays, and more variable processing delays than those of dedicated servers. In addition, we consider that the longer and more variable document fetch delays are due to lower computation power and longer disk I/O access latency on the consumer-level hosts.

Figure 8 shows the distributions of the six features for benign servers, traditional bots, and fast-flux bots. Clearly, fast-flux bots lead to much higher magnitudes for all six features compared with the other two server categories,

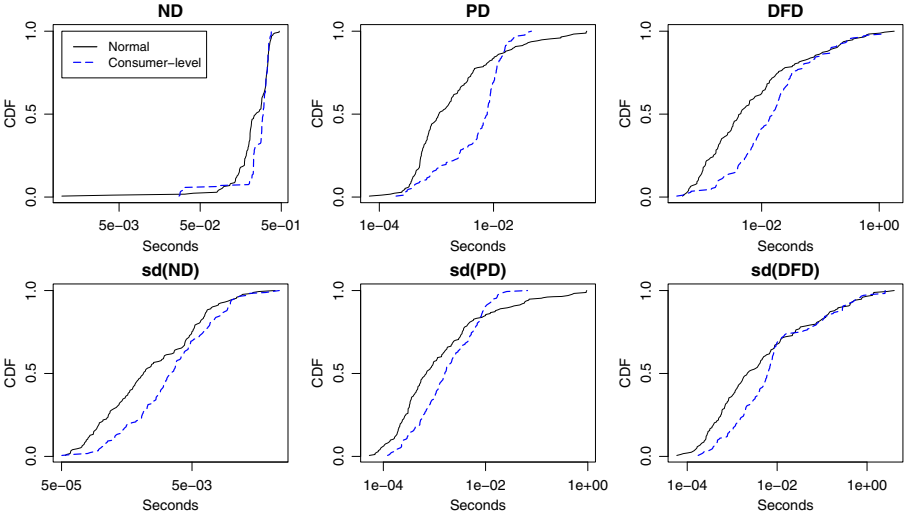


Fig. 7. The cumulative distribution functions of network delays, processing delays, and document fetch delays, and their respective standard deviations of normal and consumer-level hosts were measured based on 5 probes.

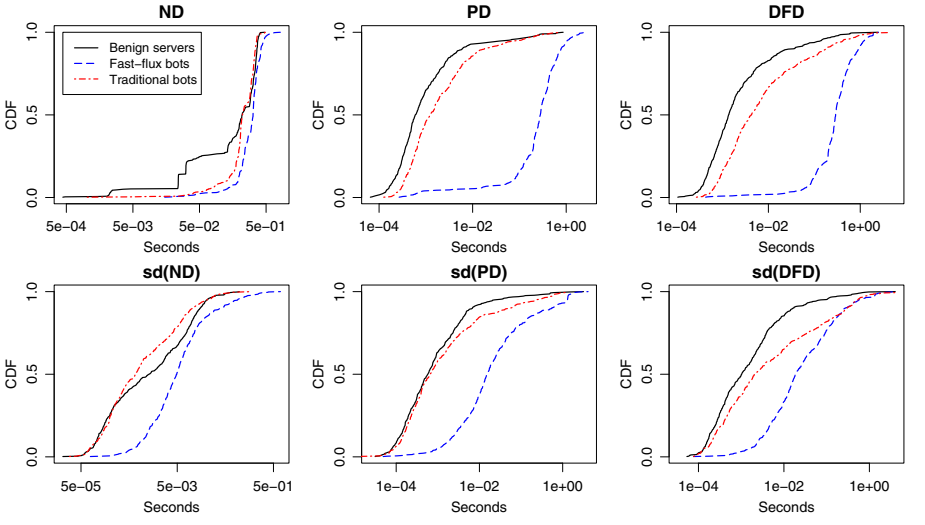


Fig. 8. The cumulative distribution functions of network delays, processing delays, and document fetch delays, and their respective standard deviations of three server categories were measured based on 5 probes

manifesting the effects of the intrinsic characteristics of fast-flux bots. The magnitudes of the six features of traditional bots are generally lower than those of fast-flux bots, but higher than those of benign servers except for the standard deviation of network delays. We believe this is because benign servers usually have more visitors than the other two categories of servers; therefore, network links to benign servers tend to be busy and it is more likely that a slightly higher degree of network queuing and delay variations will be observed.

5.3 Detection Performance

The graphs in Figure 8 confirm that the six features we derived may vary significantly according to the type of web server a user browses. In this sub-section, we perform supervised classification using SVM based on the derived six features.

Although we focus on the detection of fast-flux bots, we also include traditional bots in our evaluation. This is because, according to our analysis (Section 3), traditional bots also behave differently to benign servers in terms of most of the defined delay metrics. We perform two types of binary classification using SVM, namely, benign servers vs. fast-flux bots and benign servers vs. traditional bots. Figure 9(a) shows the relationship between the classification accuracy and the number of samples observed (which may vary according to the number of accessory files of webpages), where the accuracy is derived using 10-fold cross validation. The results show that our scheme achieves more than 95% accuracy when we try to distinguish fast-flux bots from benign servers, even when only one sample (i.e., the TCP connection) is observed. We find that it is more

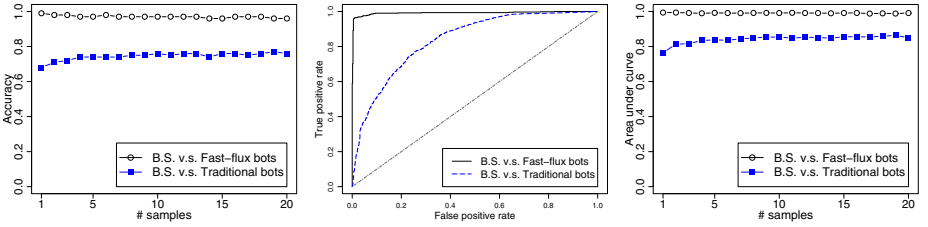


Fig. 9. (a) The relationship between the classification accuracy and the number of samples; (b) the ROC curve of the SVM classifier using 5 probes; and (c) the relationship between the area under the curve and the number of probes.

difficult to distinguish between benign servers and traditional bots because the classification accuracy is only 70%–80%; however, the accuracy rate increases when more samples are observed.

Figure 9(b) shows the ROC curves of the two types of classification based on 5 samples. The area under the curve (AUC), which distinguishes between benign servers and fast-flux bots, is 0.993; hence, the proposed detection scheme performs almost perfectly in this scenario. The AUC degrades to 0.83 when we try to classify traditional bots from benign servers, which implies that our detection scheme can detect traditional bots with a moderate degree of accuracy. As the number of samples may affect the classification performance, we plot the relationship between the AUC and the number of samples in Fig. 9(c). The graph shows that the detection performance remains nearly constant regardless of the number of samples used for fast-flux bot detection (the AUC is always higher than 0.99). In contrast, the number of samples is more important when we try to detect traditional bots, as the AUC increases above 0.8 if more than 10 samples are observed before classification is performed.

5.4 Passive Mode

The network delay and document fetch delay can be measured by passive measurements when users are browsing webpages, but an active approach must be used to measure the processing delay (i.e., by sending HTTP requests with an undefined method). Since active measurements incur additional overhead, to keep our method lightweight whenever possible, we consider that a "passive mode" would be quite useful when traffic overhead is a major concern.

In the passive mode, instead of using all six features, we only use the average and standard deviations of network delays and document fetch delays in the supervised classification. The classification accuracy is plotted in Fig. 10. We observe that the classification between fast-flux bots and benign servers is hardly affected by the removal of the "active features," i.e., processing delays and their standard deviations, except when the number of samples is quite small. We believe this indicating that document fetch delays already serve as a powerful indicator for distinguishing the two server categories. On the other hand, the

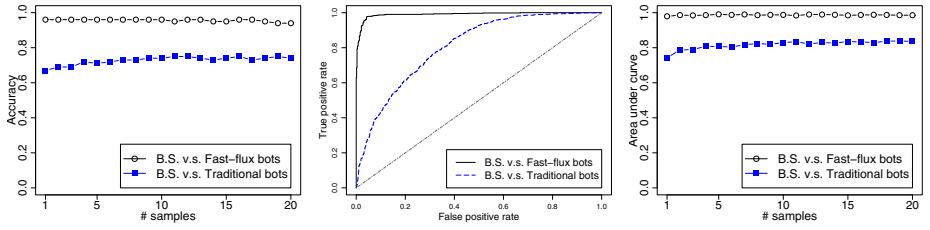


Fig. 10. (a) The relationship between the classification accuracy and the number of samples, (b) the ROC curve of the SVM classifier using 5 probes, and (c) the relationship between the area under the curve and the number of probes in the passive mode.

classification accuracy between benign servers and traditional bots is slightly affected by the removal of active features, as processing delays play an important role in distinguishing between the two types of servers. The ROC curves and the AUCs of different numbers of samples shown in Figure 10 also indicate that the passive mode of our scheme yields accurate detection results, especially when a fast-flux-bot detection method is required.

6 Discussion

In this section, we discuss several issues that are worth investigating further.

6.1 Content Delivery Network

One concern raised in a previous work on fast-flux bot detection [17] is that content delivery networks (CDNs) share a similar property with fast-flux botnets; that is, the nodes in CDNs and fast-flux botnets are associated with multiple IP addresses rather than a single IP address. This leads to confusion if a fast-flux botnet detection scheme is based on a number of IP addresses (or autonomous systems) that are associated with a certain domain name [25,17,12,3]. However, this is not a problem in our proposed method because it does not count the number of IP addresses.

6.2 Proxy Server

Although proxy servers also employ the request delegation model, we argue that the proposed scheme does not confuse fast-flux bots with proxy servers. The reason is that proxy servers are clearly visible to the end users, and the users' clients are aware that they are fetching web documents from a web server with the help of a proxy server. On the other hand, a fast-flux bot does not pretend to be a proxy server because the HTTP proxy protocol does not hide the identity of back-end web servers unless a transparent proxy is used; therefore, the mothership nodes will be revealed, which is a situation that bot herders strive

to avoid. Furthermore, if a transparent proxy is used, the proposed method will not be affected because the roles in the request delegation model are different. This is because the suspect servers contacted by users do not delegate requests to others; instead, the request-delegation operation is performed by a hidden man-in-the-middle (i.e., a transparent proxy server), which may only reduce document fetch delays. Therefore, proxy servers along the paths between users and suspect servers will not be detected as fast-flux bots.

6.3 Deployment

Our scheme can be deployed in a number of approaches. First, because of its lightweightness, it can run on end-users' machines, such as personal computers or even mobile devices. In this case, it can be implemented as a browser add-on or stand-alone software that monitors users' web browsing activities and warns users when they are browsing a website hosted by fast-flux bots.

Second, it may be more convenient if the scheme is deployed at a gateway router to protect all the users in a local area network. Since the transmission latency between a gateway router and a host is usually negligible, the delay metrics measured on the router would be roughly the same as those measured on users' computers. Therefore, we can simply monitor all outgoing HTTP requests, measure the delays, and notify users if the measurements indicate that a certain HTTP request has been sent to a fast-flux bot. We consider this to be an efficient way to deploy the proposed detection scheme to protect legitimate users.

6.4 Limitations

Although the proposed detection scheme achieves high accuracy, as shown by the results in Section 5, it has some limitations. Recall that fast-flux bots are normally equipped with consumer-level hardware and connect to the Internet with (relatively) narrower network links. The proposed scheme may fail in the following cases:

1. A bot herder may compromise powerful servers and incorporate them into a fast-flux botnet.
2. A benign server may not be equipped with high-level hardware like the dedicated web servers provided by Internet service providers.

In the first case, we believe that bots with consumer-level hardware would still dominate because high-level and high-connectivity servers are normally well-maintained and patched; hence, they are less likely to be infected and controlled by malicious software. If this should happen, we would observe short processing delays at the suspect server. The second case may occur when web servers are set up for amateur and casual use. Then, we would observe long and variable processing delays and network delays when users access webpages via such web servers. In both cases, as our method relies on all three intrinsic characteristics in the active mode (or two in the passive mode) rather than a single characteristic, a compromised server could still be detected using other characteristics, especially the "long document fetch delay" property.

7 Conclusion

We have proposed a novel scheme for detecting whether a web service is hosted by a fast-flux botnet in real time. Evaluations show that the proposed solution achieves a high detection rate and low error rates. Unlike previous approaches, our scheme does not assume that a fast-flux botnet owns a large number of bots (IP addresses). Thus, even if a botnet only owns a few bots, as long as it adopts the "request delegation" architecture, the proposed scheme can detect the botnet without any performance degradation.

In addition to being efficient and robust, the proposed solution is lightweight in terms of storage and computation costs. Therefore, it can be deployed on either fully fledged personal computers or resource-constrained devices to provide Internet users with complete protection from botnet-hosted malicious services.

Acknowledgment

This research was supported in part by National Science Council under the grant NSC 97-2218-E-019-004-MY2 and by Taiwan Information Security Center at NTUST (TWISC@NTUST) under the grant NSC 99-2219-E-011-004.

References

1. Alexa: Alexa the web information company, `http://www.alexa.com`
2. ATLAS: Arbor networks, inc., `http://atlas.arbor.net/`
3. Caglayan, A., Toothaker, M., Drapeau, D., Burke, D., Eaton, G.: Real-time detection of fast flux service networks. In: Proceedings of the Cybersecurity Applications & Technology Conference for Homeland Security, pp. 285–292 (2009)
4. Chang, C., Lin, C.: Libsvm: a library for support vector machines (2001)
5. Click Forensics, I.: Botnets accounted for 42.6 percent of all click fraud in Q3 2009 (2009), `http://www-staging.clickforensics.com/newsroom/press-releases/146-botnets-accounted.html`
6. dnsbl.abuse.ch: abuse.ch fastflux tracker (2010), `http://dnsbl.abuse.ch/fastfluxtracker.php`
7. FBI: Over 1 million potential victims of botnet cyber crime (2007), `http://www.fbi.gov/pressrel/pressrel07/botnet061307.htm`
8. Gartner: Gartner survey shows phishing attacks escalated in 2007; more than $3 billion lost to these attacks (2007), `http://www.gartner.com/it/page.jsp?id=565125`
9. Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In: Proceedings of the 17th USENIX Security Symposium (2008)
10. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting malware infection through IDS-driven dialog correlation. In: Proceedings of the 16th USENIX Security Symposium, pp. 167–182 (2007)
11. Gu, G., Zhang, J., Lee, W.: BotSniffer: Detecting botnet command and control channels in network traffic. In: Proceedings of the 15th Annual Network and Distributed System Security Symposium (2008)

12. Holz, T., Gorecki, C., Rieck, K., Freiling, F.: Measuring and detecting fast-flux service networks. In: Proceedings of the Network & Distributed System Security Symposium (2008)
13. Ianelli, N., Hackworth, A.: Botnets as a vehicle for online crime. CERT Coordination Center (2005)
14. McGrath, D., Kalafut, A., Gupta, M.: Phishing infrastructure fluxes all the way. IEEE Security & Privacy, 21–28 (2009)
15. Moore, T., Clayton, R.: Examining the impact of website take-down on phishing. In: Proceedings of the Anti-Phishing Working Groups 2nd Annual eCrime Researchers Summit (2007)
16. Namestnikov, Y.: The economics of Botnets (2009)
17. Nazario, J., Holz, T.: As the net churns: Fast-flux botnet observations. In: International Conference on Malicious and Unwanted Software, MALWARE (2008)
18. Passerini, E., Paleari, R., Martignoni, L., Bruschi, D.: FluxOR: detecting and monitoring fast-flux service networks. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 186–206 (2008)
19. PhishTank, `http://www.phishtank.com`
20. Shadowserver, `http://www.shadowserver.org`
21. Spamhaus, `http://www.spamhaus.org`
22. The Honeynet Project: Know your enemy: Fast-flux service networks (2007)
23. The Honeynet Project: Know your enemy: Tracking botnets (2008)
24. TRACELabs, M.: Marshal8e6 security threats: Email and web threats (2009)
25. Zhou, C., Leckie, C., Karunasekera, S., Peng, T.: A self-healing, self-protecting collaborative intrusion detection architecture to trace-back fast-flux phishing domains. In: Proceedings of the 2nd IEEE Workshop on Autonomic Communication and Network Management (2008)

A Client-Based and Server-Enhanced Defense Mechanism for Cross-Site Request Forgery*

Luyi Xing, Yuqing Zhang**, and Shenlong Chen

National Computer Network Intrusion Protection Center, GUCAS, Beijing 100049, China
Tel.: +86-10-88256218; Fax: +86-10-88256218
zhangyq@gucas.ac.cn
State Key Laboratory of Information Security, GUCAS, Beijing 100049, China

A common-sense CSRF attack involves more than one domain. In this paper, we'll cover both cross-domain and same-domain CSRF which overlaps with Cross-Site Scripting (XSS). If a XSS instructs victims to send requests to the same domain, it is also a CSRF—same-domain CSRF. Such sort of XSS-CSRF exists extensively and even high profile sites cannot always avoid such vulnerabilities.

There exist mainly 3 defenses: Referer Header checking, secret validation token and CAPTCHA. The Referer Header is sometimes missing [1], the secret token becomes totally futile when XSS exists and the CAPTCHA is too bothering. Besides, [2-3] brings about some client-taking actions yet pure client checking is not credible enough from server side perspective. And they still suffer from the Referer-missing problem. Moreover, all of [1-3] have nothing to do with same-domain CSRF. So a client-initialized and server-accomplished defense mechanism (CSDM) is proposed.

Definition: The **super-referer** of a request is made up of its Referer and all URLs of the Referer's ancestor frames, excluding the querying part. E.g., the Referer http://site/1.php?id=123 is cut to http://site/1.php.

CSDM proposes a new HTTP Header Super-referer-header, containing super-referer. E.g.: Super-referer-header: http://site1/index.php, http://hack/attack.aspx. Considering privacy, the URL in the new Header should be hashed with strong one-way algorithm and MD5 is one choice.

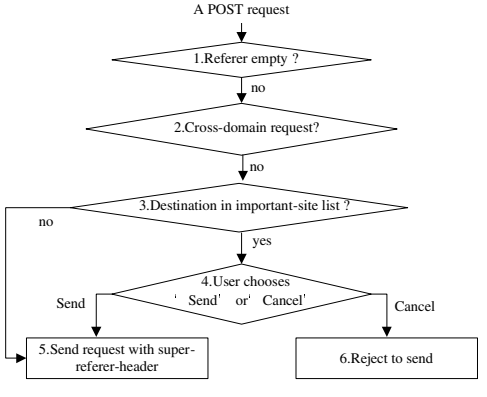


Fig. 1. Client checking of POST Request

Consider POST-based CSRF first. The client defence is shown in Fig. 1. A POST request must satisfy all the qualifications in Fig.1 before being sent out, or else it will be cancelled. In step 3 and 4, a configurable "important-sites list" is proposed. POST requests sending to important sites for users can be further confirmed by offering users a "Send or Cancel?" dialog. CSRF requests are generally sent silently and users have no idea of

* This work is supported by the National Natural Science Foundation of China under Grant No. 60970140, No.60773135 and No.90718007.
** Corresponding author.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 484–485, 2010.

it. If users didn't click any submitting button before seeing the confirming dialog, "Cancel" is preferred.

An important observation shows that POST target URL generally needs only a small number of different intended source URLs, so a policy file is used at server side. For example, `POST: {`

```
Dest1: /profile.php
Same Domain1: /chgpfl.php
Cross Domain1: trust.com/chg.aspx
Dest2: /blog.php
Same Domain2: subdomain.sns.com/*}
```

Requests sending to profile.php should only origin from chgpfl.php or trust.com/chg.aspx. The policy file should cover all POST target pages. So when addressing a request, servers examine the super-referer-header, checking whether all source URLs of the request are allowable. The server solution is deployed as part of a web application firewall, making it compatible with current websites. And we can trade space for time when decoding the MD5 value as every site has limited URLs (excluding querying parts).

Cross-domain and almost all same-domain CSRF can be prevented as their source URLs are illegal. In step 4 of Fig. 1, even attackers trigger a malicious script in parallel to the submitting of a legitimate form or the users cannot make right decisions when choosing "Send" or "Cancel", a further checking at server side will still guarantee the security. Same-domain CSRF can only happen when an allowable source page towards a specific CSRF target page happens to host some XSS vulnerability. But the chances are low and the destructiveness can be expected to be minimized or limited as only the specific target page and no others can be aimed at.

The super-referer is helpful in accurately depicting the sources of requests and preventing same-domain CSRF, as attackers can embed some permissible page in XSS-infected pages. Besides, such a concept can help preventing clickjacking [2].

GET-based CSRF deserves less attention, as all state-modifying requests should use POST and real world GET CSRF is far less destructive. At client, GET requests with HTTPS or Authorization Header are blocked if without Referrer. At server side, super-referer checking is used for sensitive target URL.

The CSDM client prototype is implemented as a Firefox browser extension. Real world tests with popular sites including iGoogle, yahoo, facebook and a vulnerable sample site show that it prevents all kinds of CSRF attacks reproduced in lab environment with no obvious compatibility problems or user experience degradation.

References

1. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: 15th ACM Conference on Computer and Communications Security (2008)
2. Mao, Z., Li, N., Molloy, I.: Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In: 13th International Conference on Financial Cryptography and Data Security (2009)
3. Maes, W., Heyman, T., Desmet, L., et al.: Browser protection against cross-site request forgery. In: 1st ACM Workshop on Secure Execution of Untrusted Code, Co-located with the 16th ACM Computer and Communications Security Conference (2009)

A Distributed Honeynet at KFUPM: A Case Study

Mohammed Sqalli, Raed AlShaikh, and Ezzat Ahmed

Department of Computer Engineering
King Fahd University of Petroleum and Minerals
{sqalli,g199607190,g200804300}@kfupm.edu.sa

1 Introduction and Design Setup

The main objectives of this work is to present our preliminary experience in simulating a virtual distributed honeynet environment at King Fahd University of Petroleum and Minerals (KFUPM) using Honeywall CDROM [1], Snort, Sebek and Tcpreplay [3] tools. In our honeynet design, we utilized the Honeywall CDROM to act as a centralized logging center for our distributed high-interaction honeypots. All honeypot servers, as well as the Honeywall CDROM itself, were built on top of a virtualized VMWare environment, while their logs were forwarded to the centralized server. This setup is illustrated in figure 1.

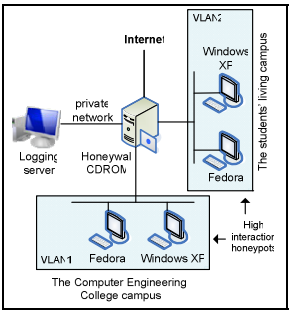


Fig. 1. The proposed distributed design of KFUPM honeynet

2 Preliminary Evaluation and Results

Since honeypots do not offer any useful services to Internet users and the Internet addresses of the honeypots are not publicly known, most traffic on the honeynet is suspicious. However, not all traffic is malicious. Therefore, the traffic we observed on our honeypots falls into three different categories:

- Network scans by KFUPM Information Technology Center.
- Traffic generated by honeypots due to normal network operations (e.g. traffic to maintain the network connection).
- Network broadcasts, such as BitTorrent requests.

At KFUPM, more than 30,000 activities were captured in the given 30-hours interval. Tale 1 shows the distribution of these types of activities in more details.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 486–487, 2010.

Table 1. The traffic distribution as it was detected by KFUPM honeynet in a 30-hours interval

Name	Protocol	Severity	Total
IIS view script source code vulnerability attack	TCP	Medium	8
MS Uni Plug and Play UDP	UDP	Medium	30
NBT(NetBIOS) Datagram Service	UDP	Low	399
Bit Torrent requests	TCP	Medium	19098
DHCP requests	UDP	Low	9938

In terms of severity, around 65% of the traffic was considered medium risk, while the remaining 35% was considered low. The high percentage of the medium-level category was due to the fact that the system classifies BitTorrents file sharing, which makes around 70% of the total traffic, as medium risk. This percentage is of no surprise since BitTorrent accounts for an astounding 40-55% of all the traffic on the Internet [5], and it is expected to be high in the students' living campuses.

Another interesting finding is the detection of a vulnerability attack on the Internet Information Service (IIS) that was installed on the Windows-based honeypots. This vulnerability has the signature KFAGC165421, and indicates that IIS contains a flaw that allows an attacker to cause IIS to return the source code for a script file instead of processing the script. This vulnerability attack traffic was generated by one of the systems in the students' living campus.

3 Conclusion and Future Work

Our experience so far shows that Honeywall CDROM proved to be a solid tool that is capable of capturing great deal of information and assisting in analyzing traffic on the distributed honeypots. The honeynet designer, nevertheless, needs to consider few issues related to scalability and resource utilization.

Out future work includes expanding our honeynet network to include other colleges and campuses in the university and have wider honeynet coverage. This will also require increasing our logging disk space to allow for more logging time, longer logging intervals and thus broader analysis.

References

1. The Honeywall CDROM, `https://projects.honeynet.org/honeywall/`
2. Argus: The Network Activity Auditing Tool, `http://www.qosient.com/argus`
3. TCPreplay, `http://tcpreplay.synfin.net/`
4. WireShark, `http://www.wireshark.org/`
5. Le Blond, S., Legout, A., Dabbous, W.: Reducing BitTorrent Traffic at the Internet Scale. A Presentation at the Internet Research Task Force, IRTF (March 2010)

Aspect-Based Attack Detection in Large-Scale Networks

Martin Drašar, Jan Vykopal, Radek Krejčí, and Pavel Čeleda

Masaryk University, Botanická 68a, 61200, Brno, Czech Republic
<surname>@ics.muni.cz

Abstract. In this paper, a novel behavioral method for detection of attacks on a network is presented. The main idea is to decompose a traffic into smaller subsets that are analyzed separately using various mechanisms. After analyses are performed, results are correlated and attacks are detected. Both the decomposition and chosen analytical mechanisms make this method highly parallelizable. The correlation mechanism allows to take into account results of detection methods beside the aspect-based detection.

1 Introduction

With the advent of multigigabit networks, the task to detect attacks becomes more and more challenging. The deep packet inspection and the pattern matching are reaching their limits. Not only hardware requirements are becoming more demanding than what can be supplied. Also a steady influx of new attacks makes all signature sets outdated by the time they are released. This situation actuated a development of new signature-less attack detection methods, namely the network behavioral analysis. The principle of such analysis is to reveal a potentially harmless behavior of network hosts by detecting deviations in traffic patterns. These methods are e. g., based on the Holt-Winters method [1] or the principal component analysis [2] and are successful in a detection of both existing and previously unknown attacks.

In the following text a new behavioral method called the aspect-based detection that offers a high speed and an ability to detect new attacks is proposed.

2 Aspect-Based Detection

Various network devices like switches or probes are able to store a set of traffic descriptors, like IP addresses, ports and transferred data for each connection. A combination of one or more of these descriptors represents one aspect of traffic, e. g., i) amount of traffic from one address (source address, payload size, time) or ii) traffic volume on given port over the time (destination port, payload size, time).

Every aspect can be represented as a multidimensional matrix where each element stands for one connection. These matrices are subject to analysis. This

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 488–489, 2010.

approach has several advantages. First, by splitting the entire traffic into its aspects the volume of data that has to be processed at one time is lowered, thus relaxing hardware requirements. Aspects also conserve relations in traffic patterns. Second, by representing traffic data as a matrix, fast and specialized algorithms from the area of the digital signal processing can be used.

The core operation in the aspect-based detection is an application of linear and non-linear filters on aspect matrices. Non-linear filters are mainly used for thresholding – representing static anomaly checks, e. g., one computer connects to a hundred other computers. Linear filters are used to tackle dynamic aspects of a traffic. A convolution with Sobel-like operators that approximate the second derivation can effectively discover sudden changes in traffic, that can e. g., point to an activation of infected computer. Aspect matrices or the entire traffic can also be fed to other detection mechanisms, provided their result can be converted to a matrix with values comparable to other transformed aspect matrices.

Each transformed aspect matrix can identify an ongoing attack, but it is more likely to only highlight traffic deviations. To discover more stealthy attacks, it is necessary to correlate these matrices. This is done by constructing a resulting matrix which dimensions are sum of dimensions of transformed aspect matrices. Individual matrices are added to the resulting one in a manner that influences also dimensions the added matrix is not defined for. This addition is best described by an example. Let $R_{a,b,c,d}$ be a four-dimensional resulting matrix and $A_{a,c,d}$, $B_{a,b}$ and $C_{c,d}$ be transformed aspect matrices. A calculation of one element goes like this: $R_{a_x,b_y,c_z,d_w} = A_{a_x,c_z,d_w} + B_{a_x,b_y} + C_{c_z,d_w}$.

The resulting matrix describes traffic in terms of identified deviations. There are likely to be three kinds of areas in this matrix – where a deviation going over certain thresholds i) indicates an attack, ii) is harmless, iii) is suspicious. These thresholds have to be hand selected at least for known attacks. But for unknown attacks these thresholds might be derived e. g., from a distance from a known attack in the resulting matrix. Distance metrics are to be modified according to a collected data.

3 Future Work

The research of the aspect-based detection has to focus on several key areas. Main task is to create appropriate data structures that will allow effective processing of aspect matrices. Also the previously mentioned areas in the resulting matrix must be identified and metric-derived thresholds investigated.

References

1. Li, Z., Gaoa, Y., Chen, Y.: HiFIND: A high-speed flow-level intrusion detection approach with DoS resiliency. Computer Networks 54(8), 1282–1299 (2010)
2. Lakhina, A., Crovella, M., Diot, C.: Anomaly Detection via Over-Sampling Principal Component Analysis Studies. Computational Intelligence 199, 449–458 (2009)

Detecting Network Anomalies in Backbone Networks

Christian Callegari, Loris Gazzarrini, Stefano Giordano,
Michele Pagano, and Teresa Pepe

Dept. of Information Engineering, University of Pisa, Italy
{c.callegari,l.gazzarrini,s.giordano,m.pagano,t.pepe}@iet.unipi.it

1 Extended Abstract

The increasing number of network attacks causes growing problems for network operators and users. Thus, detecting anomalous traffic is of primary interest in IP networks management. As it appears clearly, the problem becomes even more challenging when taking into consideration backbone networks that add strict constraints in terms of performance.

In recent years, Principal Component Analysis (PCA) has emerged as a very promising technique for detecting a wide variety of network anomalies. PCA is a dimensionality-reduction technique that allows the reduction of the dataset dimensionality (number of variables), while retaining most of the original variability in the data. The set of the original data is projected onto new axes, called Principal Components (PCs). Each PC has the property that it points in the direction of maximum variance remaining in the data, given the variance already accounted for in the preceding components.

In this work, we have focused on the development of an anomaly based Network Intrusion Detection System (IDS) based on PCA. The starting point for our work is represented by the work by Lakhina et al. [1], [2]. Indeed, we have taken the main idea of using the PCA to decompose the traffic variations into their normal and anomalous components, thus revealing an anomaly if the anomalous components exceed an appropriate threshold. Nevertheless, our approach introduces several novelties in the method, allowing great improvements in the system performance. First of all we have worked on four distinct levels of aggregation, namely ingress router, origin-destination flows, input link, and random aggregation performed by means of sketches, so as to detect anomalies that could be masked at some aggregation level. In this framework, we have also introduced a novel method for identifying the anomalous flows inside the aggregates, once an anomaly has been detected. To be noted that previous works are only able to detect the anomalous aggregate, without providing any information at the flow level. Moreover, in our system PCA is applied at different time-scales. In this way the system is able to detect both sudden anomalies (e.g. bursty anomalies) and "slow" anomalies (e.g. increasing rate anomalies), which cannot be revealed at a single time-scale. Finally, we have applied, together with the entropy, the Kullback-Leibler divergence for detecting anomalous behavior, showing that our

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 490–491, 2010.

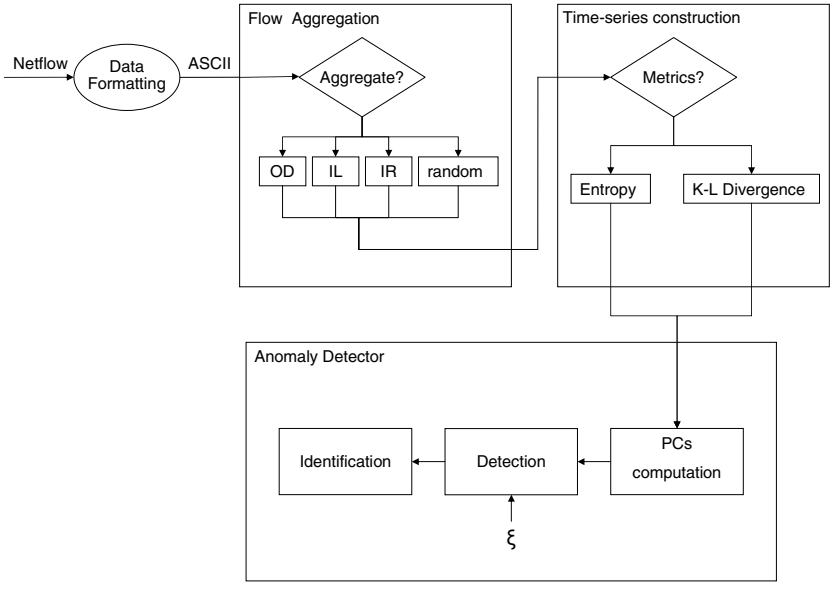


Fig. 1. System Architecture

choice results in better performance and more stability for the system. Figure 1, shows the architecture of the proposed system.

The proposed system has been tested using a publicly available data-set, composed of traffic traces collected in the Abilene/Internet2 Network [3], that is a hybrid optical and packet network used by the U.S. research and education community. Since the data provided by the Internet2 project do not have a ground truth file, we are not capable of saying *a priori* if any anomaly is present in the data. For this reason we have partially performed a manual verification of the data, analyzing the traces for which our system reveals the biggest anomalies. Moreover we have synthetically added some anomalies in the data (mainly representative of DoS and DDoS attacks), so as to be able to correctly interpret the offered results, at least partially. The performance analysis has highlighted that the implemented system obtains very good results, detecting all the synthetic anomalies.

References

1. Lakhina, A., Crovella, M., Diot, C.: Characterization of network-wide anomalies in traffic flows. In: ACM Internet Measurement Conference, pp. 201–206 (2004)
2. Lakhina, A.: Diagnosing network-wide traffic anomalies. In: ACM SIGCOMM, pp. 219–230 (2004)
3. The Internet2 Network, `http://www.internet2.edu/network/`

Detecting the Onset of Infection for Secure Hosts

Kui Xu[1], Qiang Ma[2], and Danfeng (Daphne) Yao[1]

[1] Department of Computer Science, Virginia Tech
{xmenxk,danfeng}@cs.vt.edu
[2] Department of Computer Science, Rutgers University
qma@cs.rutgers.edu

Abstract. Software flaws in applications such as a browser may be exploited by attackers to launch drive-by-download (DBD), which has become the major vector of malware infection. We describe a host-based detection approach against DBDs by correlating the behaviors of human-user related to file systems. Our approach involves capturing keyboard and mouse inputs of a user, and correlating these input events to file-downloading events. We describe a real-time monitoring system called *DeWare* that is capable of accurately detecting the onset of malware infection by identifying the illegal download-and-execute patterns.

Analysis based on the arrival methods of top 100 malware infecting the most number of systems discovered that 53% of infections are through download [1]. In another study, 450,000 out of 4.5 millions URLs were found to contain drive-by-download exploits that may be due to advertisement, third-party contents, and user-contributed contents [2]. Drive-by-download (DBD) attacks exploit the vulnerabilities in a browser or its external components to stealthily fetch malicious executables from remote malware-hosting server without proper permission of the user.

We present *DeWare* – a host-based security tool for detecting the onset of malware infection at real time, especially drive-by-download attacks. Deware is application independent, thus it is capable of performing host-wide monitoring beyond the browser. DeWare's detection is based on observing stealthy download-and-execute pattern, which is a behavior virtually all active malware exibits at its onset.

However, the main technical challenge to successful DBD detection is to tell DBDs apart from legal downloads. Our solution is based on monitoring relevant file-system events and correlating them with user inputs at the kernel level. In contrast to DBDs, legitimate user download activities are triggered by explicit user requests. Also, browser itself may automatically fetch and create temporary files which are not directly associated with user actions. To that end, we grant browser access to limited folders with additional restrictions.

Security and attack models. We assume that the browser and its components are not secure and may have software vulnerabilities. The operating system is assumed to be trusted and secure, and thus the kernel-level monitoring of

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 492–493, 2010.

file-system events and user inputs yields trusted information. The integrity of file systems defined in our model refers to the enforcement of user-intended or user-authorized file-system activities; the detection and prevention of malware-initiated tampering.

DeWare Archietecture Overview. The DeWare monitoring system is designed to utilize a combination of three techniques, including input logger, system monitor, and execution monitor. Following are the main components.

- *Input logger* that intercepts user inputs at the kernel level with timestamp and process information (i.e., to which process the inputs go to). User inputs are viewed as *trusted seeds* in our analysis, which are used to identify legitimate system behaviors.
- *System logger* which intercepts system calls for file creations, and probes kernel data structures to gather process information. Timestamps can be obtained from input logger at runtime to perform temporal correlation.
- *Access control framework* that specifies (1)accessible area: where an application is allowed to make file creations, (2)downloadable area: places a user can download files into via an application.
- *Execution monitor* which gives additional inspection to areas where access is granted to an application or user downloads.

Capturing all file-creation events related to processes generates an overwhelmingly large number of false alarms. The purpose of our access control framework is to reduce the white noise, by granting a process access to certain folders, which are defined as accessible area. For example, Temporary Internet Files folder is modifiable by IE – in contrast, system folder is not. *Execution monitor* is to prevent malware executables from being run at accessible area.

Prototype Implementation in Windows Our implementation and experiments are built with Minispy, a kernel driver for Windows operating systems. It is able to monitor all events where system is requesting to open a handle to a file object or device object, and further find out the file creations. Logged activities are reported to user mode where the access control policy, input correlation, file extension check are performed. We record user inputs at the kernel level through hooks SetWindowsHookex provided by Windows OS. The *execution monitor* is realized with Microsoft PsTools and the process tracking in local security settings. We have carried out a study with 22 users to collect real-world user download behavior data. We will also use DeWare to evaluate a large number of both legitimate and malware-hosting websites for testing its detection accuracy.

References

1. Macky Cruz. Most Abused Infection Vector, `http://blog.trendmicro.com/most-abused-infection-vector/`
2. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The ghost in the browser analysis of web-based malware. In: Hot-Bots 2007: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets. USENIX Association, Berkeley (2007)

Eliminating Human Specification in Static Analysis*

Ying Kong, Yuqing Zhang**, and Qixu Liu

National Computer Network Intrusion Protection Center, GUCAS, Beijing 100049, China
Tel.: +86-10-88256218; Fax: +86-10-88256218
zhangyq@gucas.ac.cn
State Key Laboratory of Information Security, GUCAS, Beijing 100049, China

Abstract. We present a totally automatic static analysis approach for detecting code injection vulnerabilities in web applications on top of JSP/servlet framework. Our approach incorporates origin and destination information of data passing in information flows, and developer's beliefs on vulnerable information flows extracted via statistical analysis and pattern recognition technique, to infer specifications for flaws without any human participation. According to experiment, our algorithm is proved to be able to cover the most comprehensive range of attack vectors and lessen the manual labor greatly.

Published static approaches for detecting code injection vulnerabilities depend on human work heavily to specify flaws and to build auditing model. This leads to much omission in tagging attack vectors due to the erratic nature of human judgment, furthermore, the omission in flaw report. In this paper, we present a novel approach named injection vulnerability checking tool (IVCT) to solve this problem.

We consider the attack against code injection vulnerability as an improper communication procedure among three components including the front-end web server, the back-end database and the underlying operating system. Return from method invoked on web server forms the message, and is finally accepted by another method invoked on one of the three components. We treat the former method as taint source, and the latter as vulnerable receiver. Data flow of the message, which starts with taint source and ends at vulnerable receiver, is regarded as possible candidate of vulnerable flow in this paper. Such model covers the most comprehensive range of attack vectors.

IVCT framework consists of four phases, which are illustrated in Fig 1. We take advantage of the slicing technique [1] described in [2] to track propagation of untrusted input, and enhance the dataflow analysis with **indirect propagation** which models the relationship between the data passing into and out of a library method and abstracts away the concrete operations on data therein. Such abstraction is based on the insight that most library code won't modify data structure from customer code. Before tracking, just those sensitive components' jar paths are required be specified in advance to locate the candidate information flows. During tracking, we can collect tainted information propagated via library invocation directly instead of tracking into the implementation. For example, in the statement "str2=a.fun(str1)", data "str1"

* This work is supported by the National Natural Science Foundation of China under Grant No. 60970140, No.60773135 and No.90718007.
** Corresponding author.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 494–495, 2010.

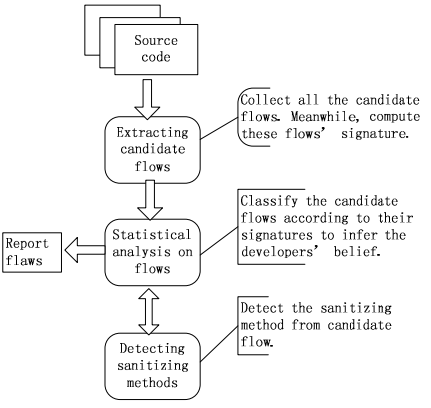


Fig. 1. IVCT Workflow

Table 1. Experimental Results Comparing with TAJ and bddbddb

Application	XSS	SQLi	DT	C I	Total	FP
bddbddb	?	?		0	6	0
TAJ	?	?	?	?	37	6
IVCT	94	24	4	3	125	7
Webgoat						
bddbddb	0	0	2	0	2	0
TAJ	?	?	?	?	127	many
IVCT	3	0	3	0	6	0
blojsom						

passes into library invocation "a.fun", then reference variable "a" and "str2" will be treated as tainted data passing out of the invocation. Such enhancement are expected to simplify the tracking process, and hence to improve the scalability.

We manually inspected two web applications "Webgoat5.3RC" and "blojsom-3.3b", both of which are used by tools TAJ in Tripp [2] and bddbddb in [3] for experiment data. In the analysis, IDE "Eclipse" is utilized to locate grammar element of java code, the rest operations are rigorously adhered to IVCT's instructions. Therefore, no human judgments have been involved into the inspection. According to the experimental results illustrated in Table 1, our approach is proved to be better in two factors. First, no human participation is required by IVCT. In contrast, TAJ and bddbddb require checkers to read the libraries used by targeted web applications thoroughly to flag taint sources and sinks. Second, IVCT captures more vulnerabilities with fewer false positives. We own the bigger number to the fact that IVCT's candidate flows cover all the attack vectors. In fact, every method returning variable possible to carry string value in web server library is potential taint source, but TAJ limits taint source only in form input and upload file data. Additionally, variables propagated by sinks' reference variables are potential vulnerable receivers. However, such propagation is ignored by both [2] and [3]. In the future, we plan to implement our approach in a tool to be used in real code. In addition, try to extract other beliefs buried in program code which can be used as flaw specification.

References

1. Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, vol. 42(6), pp. 112–122 (2007)
2. Tripp, O., Pistoia, M., Fink, S., Sridharan, M., Weisman, O.: TAJ: Effective Taint Analysis of Web Applications. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 87–97 (2009)
3. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: The 14th USENIX Security Symposium, pp. 271–286 (2005)

Evaluation of the Common Dataset Used in Anti-Malware Engineering Workshop 2009

Hosoi Takurou and Kanta Matsuura

Institute of Industrial Science, The University of Tokyo
4-6-1, Komaba, Meguro-ku, Tokyo 153-8585, Japan

Abstract. Anti-Malware Engineering Workshop 2009 provided a common dataset for all the authors there. In order to understand research-promotion effects in the network-security community, we evaluate the dataset through observations and a questionnaire.

Keywords: malware, evaluation dataset, network security.

1 Introduction

Evaluation by using datasets is a major approach in network security due to the difficulty of theoretical evaluation. If a common dataset is available, we can have more reliable comparison among different technologies. And if the dataset is better maintained, the absolute quality of each evaluation gets better. A Japanese domestic workshop on network security, called anti-Malware engineering WorkShop 2009 (MWS2009) [3], was challengingly designed in a way that all the 28 authors should use a common dataset (CCC DATAset 2009 [2]). In order to understand effects of this challenge on research promotion, we evaluate the dataset through observations and a questionnaire.

2 Observations

A well-known example of commonly available datasets is DARPA dataset [1] which is basically for intrusion-detection evaluation. By contrast, CCC DATAset 2009 has a more comprehensive nature with the following three classes of data:

(S) malware specimen information (on 10 malwares),
(T) attack traffic data (by 2 hosts, 2 days long), and
(A) attack source log (by 94 hosts, 1 year long).

These data were captured at 94 honeypots during one year from the middle of 2008 to the middle of 2009, and were provided along with the dataset used in the previous year's edition of MWS. This comprehensiveness is an advantage; the more researchers join the chellenging workshop, the higher the productivity of the challenge is.

Another remarkable feature of the dataset is operational efforts for organizing the workshop (e.g. carefully-designed contracts among stakeholders). The realization of the workshop itself and its sustainability (in fact, the Japanese community is preparing MWS2010) suggests benefits from this feature.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 496–497, 2010.

3 Questionnaire-Based Evaluation

3.1 Questionnaire

We sent a questionnaire to all the users of the dataset, and received 27 responses. The questionnaire consists of 89 questions, which are on the role of replying person and other administrative aspects (8 questions), on technical aspects in general (14 questions), on the data of class (S) (17 questions), on the data of class (T) (29 questions), and on the data of class (A) (21 questions). The large number of questions on technical aspect were designed in a systematic manner; many of them ask "expectation before use" as well as "evaluation after use", and "absolute evaluation considering their demand" as well as "comparison of the absolute evaluation with their own dataset (i.e. not the common dataset but the dataset which the researcher prepared by themselves)".

3.2 Result

Due to the page limitation, we here show some remarkable results only.

The rate of deployment: The ratio of the number of users who used each class of data to the number of users who planned to use them before starting their research are: $\langle 8/11 \rangle$ for the class (S), $\langle 17/20 \rangle$ for the class (T), and $\langle 10/13 \rangle$ for the class (A). It should be noted that the ratio is $\langle 9/16 \rangle$ for responses from researchers who used data of multiple classes. The importance of dataset comprehensiveness is thus suggested.

The usefulness of the dataset: Regarding the usefulness of the dataset of each class, the negative answers are very few: 1 out of 8 for class (S), 0 out of 17 for class (T), and 0 out of 10 for class (A). The high productivity of the project is thus suggested.

4 Concluding Remarks

Through observations and a questionnaire-based evaluation, we found that CCC DATAset 2009 has many good features and is supported by participating researchers. It is suggested that the comprehensiveness of the dataset brings a large impact. In the poster, more details will be described.

References

1. DARPA intrusion detection evaluation dataset, `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html`
2. Hatada, M., Nakatsuru, Y., Terada, M., Shinoda, Y.: Dataset for anti-malware research and research achievements shared at the workshop. In: Computer Security Symposium 2009 (CSS 2009), Anti-Malware Engineering WorkShop 2009 (MWS 2009), IPSJ, Japan, pp. 1–8 (2009) (in Japanese)
3. anti-Malware engineering WorkShop 2009 (MWS 2009), `http://www.iwsec.org/mws/2009/en.html`

Inferring Protocol State Machine from Real-World Trace

Yipeng Wang[1 2], Zhibin Zhang[1], and Li Guo[1]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Graduate University, Chinese Academy of Sciences, Beijing, China
wangyipeng@software.ict.ac.cn

Abstract. Application-level protocol specifications are helpful for network security management, including intrusion detection, intrusion prevention and detecting malicious code. However, current methods for obtaining unknown protocol specifications highly rely on manual operations, such as reverse engineering. This poster provides a novel insight into inferring a protocol state machine from real-world trace of a application. The chief feature of our method is that it has no priori knowledge of protocol format, and our technique is based on the statistical nature of the protocol specifications. We evaluate our approach with text and binary protocols, our experimental results demonstrate our proposed method has a good performance in practice.

1 Introduction and System Architecture

Finding protocol specifications is a crucial issue in network security, and detailed knowledge of a protocol specification is helpful in many network security applications, such as intrusion detection systems and vulnerability discovery *etc.* In the context of extracting protocol specifications, inferring the protocol state machine plays a more important role in practice. ScriptGen [1] is an attempt to infer protocol state machine from network traffic. However, the proposed technique is limited for no generalization.

This poster provides a novel insight into inferring a protocol state machine from real-world packet trace of an application. Moveover, we propose a system that can automatically extract protocol state machine for stateful network protocols from Internet traffic. The input to our system is real-world trace of a specific application, and the output to our system is the protocol state machine of the specific application. Furthermore, our system has the following features, (a) no knowledge of protocol format, (b) appropriate for both text and binary protocols, (c) the protocol state machine we inferred is of good quality.

The objective of our system is to infer the specifications of a protocol that is used for communication between different hosts. To this end, our system carries on the whole process in four phases, which are shown as follows:

Network data collection. In this phase, network traffic of a specific application (such as SMTP, DNS *etc.*) is collected carefully. In this poster, The method of collecting packets under specific transport layer port is adopted.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 498–499, 2010.

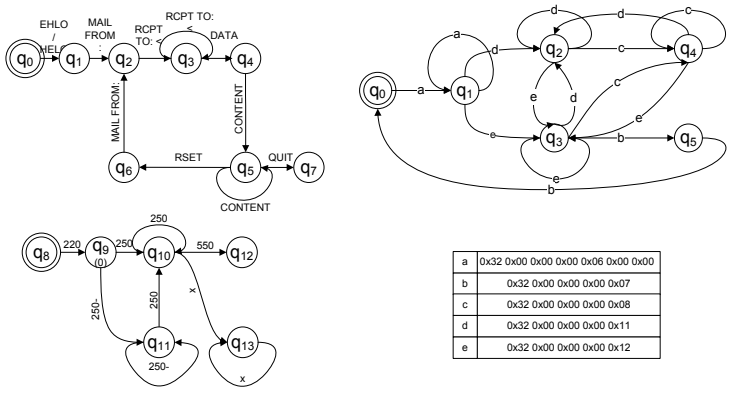


Fig. 1. The Protocol State Machine of SMTP and XUNLEI Protocol

Packet analysis. During the part of packet analysis, we first look for high frequency units from off-line application-layer packet headers, which is obtained by the phase of network data collection. Then, we employ Kolmogorov-Smirnov (K-S) test to determine the optimal number of units. Finally, we replay each application-layer packet header and construct protocol format messages with objective units.

Message clustering. In this phase, we extract the feature from each protocol format message. The feature is used to measure the similarity between messages. Then, the partitioning around medoids (PAM) clustering algorithm is applied to group similar messages into a cluster. Finally, the medoid message of a cluster will become a protocol state message.

State machine inference. In order to infer protocol state machine, we should be aware of the packet state sequence of flows. For the purpose of labeling the packet state, initially we have to find the nearest medoid message of each packet and assign the identical label type to the packet. Then, by finding the relationship between different state types, a protocol machine is constructed. After state machine minimization, we will get the ultimate protocol state machine.

2 Evaluation

We make use of SMTP (text protocol) and XUNLEI (binary protocol) to test and verify our method. The protocol state machine of SMTP we inferred is shown in Fig. 1 left, and XUNLEI in right. Moreover, our evaluation experiments show that our system is capable of parsing about 86% flows of SMTP protocol and about 90% flows of XUNLEI protocol.

Reference

1. Leita, C., Mermoud, K., Dacier, M.: Scriptgen: an automated script generation tool for honeyd. In: Annual Computer Security Applications Conference (2005)

MEDUSA: Mining Events to Detect Undesirable uSer Actions in SCADA

Dina Hadžiosmanović, Damiano Bolzoni and Pieter Hartel

Distributed and Embedded Security, University of Twente
{dina.hadziosmanovic,damiano.bolzoni,pieter.hartel}@utwente.nl

Abstract. Standard approaches for detecting malicious behaviors, e.g. monitoring network traffic, cannot address process-related threats in SCADA(Supervisory Control And Data Acquisition) systems. These threats take place when an attacker gains user access rights and performs actions which look legitimate, but which can disrupt the industrial process. We believe that it is possible to detect such behavior by analysing SCADA system logs. We present MEDUSA, an anomaly-based tool for detecting user actions that may negatively impact the system.

1 Problem

There is a wide range of possible attacks that can be carried out against SCADA environments [1,2]. We classify possible threats in two groups: system- and process-related. System-related threats are typical of "regular" computer networks, e.g., malware or Denial of Service attacks. Attackers leverage vulnerabilities in networked systems and programmable logic controllers (PLCs) to alter or disrupt the industrial process. Process-related threats imply that an attacker gains user access rights (e.g., through social engineering) and performs legitimate SCADA commands which will negatively affect the industrial processes. Typical security countermeasures, e.g., antivirus or network IDSes, can hardly detect process-related threats, as they lack process semantic.

In this work, we focus on the detection of process-related threats. Based on interviews with stakeholders, we distinguish two types of threat scenarios, namely 1) an attacker impersonates a system user or 2) a legitimate system user makes an operational mistake. A SCADA engineer manages object libraries and user interfaces, sets working ranges for devices, etc. If an attacker succeeds in acquiring the access rights of an engineer, she is then able to perform actions such as altering a device parameter (e.g., change capacity of a tank) or altering the system topology (e.g. some devices become "invisible", and thus inaccessible). A SCADA operator monitors the system status and reacts to events, such as alarms, so that the process runs correctly. An attacker, impersonating an operator or an engineer, can generate a sequence of actions where each action is legitimate, but the combination (or even a single action) can damage the process.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 500–501, 2010.

We argue that to detect process-related attacks one needs to analyse data passed through the system (Bigham et al. [1]) and include a semantical understanding of the process and user actions. This can be achieved either by employing a tool such Bro, which requires the network protocol specifications (but those could be hard to obtain due to the closeness of SCADA systems), or by analysing system logs.

2 Solution

Typically, SCADA system logs provide detailed information about industrial processes. However, based on interviews with stakeholders, logs are not normally processed. The reason for this is that system engineers lack time, skills and specific tools for performing a thorough analysis. The size and high dimensionality of the logs make manual inspection infeasible. For instance, a SCADA system for a water treatment process in a medium-size city, depending on daily activities, records between 5.000 and 15.000 events per day.

We believe that system logs can be used to detect process-related threats and user mistakes automatically. We propose a visualization tool, MEDUSA(Mining Events to Detect Undesirable uSer Actions in SCADA), whose engine is based on anomaly detection. MEDUSA automatically analyses system logs, detects and alerts users about situations in which the system behaves unconsistently with past behavior. As a result, the number of security-related alarms that operators have to look at is decreased. The anomaly detection models in MEDUSA are built using two data mining techniques. First, we use algorithms for mining outliers to detect individual actions and events that are significantly different from previous entries. Secondly, we analyse sequences of events and actions in the logs to provide a better view on the situation context. Once we train our model on history logs of a particular system, we plan to use the model in real-time analysis a SCADA system.

Preliminary results show that our approach is feasible. The initial dataset consists of 100.000 entries which correspond to approximatively 15 days of process work. The attributes are mostly categorical. The goal of our initial analysis was to transform the dataset in such a way that anomalous entries are highlighted. We managed to extract several events that may semantically represent suspicious behavior (eg., a previously unseen engineer activity in late night hours, user expression errors when connecting to critical communication backbones).

References

1. Bigham, J., Gamez, D., Lu, N.: Safeguarding scada systems with anomaly detection. In: MMMACNS 2003: Proc. 2nd International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security. LNCS, pp. 171–182. Springer, Heidelberg (2003)
2. Chittester, C.G., Haimes, Y.Y.: Risks of terrorism to information technology and to critical interdependent infrastructures. Journal of Homeland Security and Emergency Management, Article 402 1(4), 341–348 (2004)

On Estimating Cyber Adversaries' Capabilities: A Bayesian Model Approach*

Jianchun Jiang[1], Weifeng Chen[2], and Liping Ding[1]

[1] National Engineering Research Center of Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing, China
jianchun@nfs.iscas.ac.cn, dlp@iscas.ac.cn
[2] Dept. of Math and Computer Science, California University of PA, USA
chen@calu.edu

1 Introduction

Cyber adversaries refer to people or groups who do harm to the information system, such as hackers, espionage persons, and terrorists. Different Cyber adversaries have different motivations, and obviously, have different resources and attack techniques. The resource and attack techniques are referred to as *adversaries' capacities*. Accurate estimation of adversaries' capacities can help network administrator to use different approaches to prevent potential attacks or respond to emerging attacks. However, cyber adversaries' capabilities are hidden, dynamic and difficult to observe directly. This poster aims to take a systemic approach to estimate adversaries' capacities. Since we cannot obtain complete information about the adversaries, a reasonable approach is to estimate adversaries' capabilities using partial information that has been observed. The estimation hypothesis, initially stating that the adversary has equal probabilities to have high level capacities and low level capacities, will be refined using Bayesian rules as we collect more evidences from network data.

2 A Bayesian Model

We use H to represent the hypothesis "The cyber adversary's capability is high level". Based on Bayes' theorem, we can update the degree of belief of hypothesis H given an evidence E in the following way:

$$P(H|E) = \frac{P(E|H)}{P(E)} \times P(H) \tag{1}$$

* This work is supported in part by the National High-Tech Research and Development Plan of China under Grant No.2007AA010601 and the Knowledge Innovation Key Directional Program of Chinese Academy of Sciences under Grant No. KGCX2-YW-125. The authors would like to thank Dr. Peng Ning at North Carolina State University for his insightful discussion.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 502–504, 2010.

3 Extracting Evidences from Network Data

Transferring from network data to evidences plays an essential role in this refining process. We divide network data into different categories and build a database that correlates the data categories with attack scenarios. Due to the space limitation, here we only describe the "Exploit" evidence as an example. The "Exploit" category describes an adversary's characteristics of exploiting vulnerabilities of the target. Let V_d be the publication date of the vulnerability that is exploited by the adversary. Let V_c be the date when the vulnerability is exploited by the adversary. Generally, if $V_c - V_d$ is small, it means that the adversary has strong capability in exploring the target, e.g., the zero-day attack. We then use the ratio $\frac{1}{V_c - V_d}$ to represent exploit capability of the adversary.

$$\begin{cases} P(\text{Exploit}|H) \approx \frac{1}{V_c - V_d} & \text{where } V_c - V_d \neq 0 \text{ and } V_c > V_d \\ P(\text{Exploit}|H) \approx 1 & \text{otherwise} \end{cases}$$

4 Case Study

More and more Cyber adversaries are interested in attacking popular Web sites, commonly by exploring vulnerabilities of the Web sites. Based on the network

Table 1. Selected vulnerabilities and their "Exploit" values in year 2007.

Adversary	Vulnerability	V_d (dd.mm.yyyy)	V_c(dd.mm.yyyy)	$V_c - V_d$
A1	MS07-004	09.01.2007	26.01.2007	17
A2	MS07-009	24.10.2006	28.03.2007	158
A3	MS07-017	28.03.2007	30.03.2007	2
A4	MS07-020	10.04.2007	15.09.2007	155
A5	MS07-033	14.03.2007	07.07.2007	113
A6	MS07-035	12.06.2007	11.07.2007	29
A7	MS07-045	15.08.2007	02.09.2007	17
A8	CVE-2007-3148	06.06.2007	08.06.2007	2
A9	CVE-2007-4105	02.08.2007	18.08.2007	16
A10	CVE-2007-4748	19.08.2007	19.08.2007	0
A11	CVE-2007-5017	19.09.2007	26.09.2007	7
A12	CVE-2007-3296	30.05.2007	25.06.2007	25
A13	CVE-2007-5064	30.08.2007	30.08.2007	0

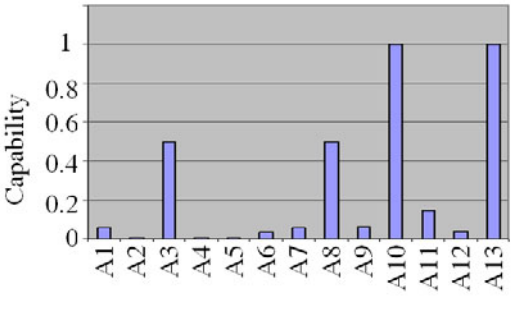


Fig. 1. Different Capabilities for Cyber Adversary Hacking Website

data about the Chinese Web sites [1] for selected vulnerabilities in the year 2007, we extract the "Exploit" evidence and use this evidence to illustrate the application of our model. Table 1 shows 13 adversaries, vulnerabilities exploited by each adversary, and their V_c and V_d.

Based on this information, we applied our model described in this poster and obtain the $P(H)$ for the 13 adversaries, as shown in Figure 1.

Reference

1. Zhuge, J., Holz, T., Song, C., Guo, J., Han, X., Zou, W.: Studying Malicious Websites and the Underground Economy on the Chinese Web. In: 7th Workshop on the Economics of Information Security (WEIS 2008), Hanover, NH, USA (June 2008)

Security System for Encrypted Environments (S2E2)

Robert Koch and Gabi Dreo Rodosek

Universität der Bundeswehr München, 85577 Neubiberg, Germany
{Robert.Koch,Gabi.Dreo}@UniBw.de

Abstract. The percentage of encrypted network traffic increases steadily not only by virtual private networks of companies but also by protocols like SSH or SSL in the private sector. Traditional intrusion detection systems (IDS) are not able to cope with encrypted traffic. There are a few systems which are able to handle encrypted lines but none of them is applicable in general because of changed network protocols, a restricted application range (e.g., only able to find protocol-specific attacks) or very high false alarm rates. We propose a new IDS for non-intrusive, behavior-based intrusion- and extrusion detection in encrypted environments.

Keywords: intrusion detection, payload encryption, non-intrusive measurement, user strategy, traffic clustering, extrusion detection, data leakage detection.

1 Background

Signature-based IDSs (misuse detection) are widely used for the protection of networks. Because patterns of the malware must be available in the database of the IDS, only already known threats can be found. A study of the Massachusetts Institute of Technology in the year 2002 unfoldes, that software patches are often available at the same time as the signatures for IDSs are [1], therefore reducing the advantages of the IDS. Even more, the increasing percentage of encrypted network traffic additionally limits the detection capability of those systems, because they have to analyze the payload and are not able to cope with encrypted traffic. Unlike the misuse detection, anomaly-based systems are able to detect new or unknown threats. E.g., the spreading of new worms can be detected, but attacks *inside* the encrypted traffic (on application layer) are still not detectable. Currently, there are only few IDSs able to cope with encrypted traffic but none of them is applicable in general. [2] gives an overview of available systems and also proposes a new one (which also has the same restrictions).

2 S2E2 System Architecture

S2E2 is an anomaly-based system. All parts of the system are working non-intrusive, a decryption is not necessary. Based on the observable encrypted

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 505–507, 2010.

network traffic, the user input is identified and weighted. Concurrently, the user generating the network traffic is identified by keystroke dynamics. The therefor necessary features are recovered by the timing of the network packets. The system architecture consists of the following modules:

Traffic Clustering: The system records the *timestamps* of the network packets, the *payload sizes* and the transmission *directions*. The gathered data is grouped into clusters, whereas a cluster consists of an user input and the corresponding answer of the server.

Command Evaluation: This is done by analysing the *consecutive payload sizes* of the network packets. Timestamps are taken into consideration as well, e.g. for the detection of server delays (for example, the delay when requesting the listing of a directory is all the longer with the number of files in the directory). In the first step, probabilities for single command-answer-combinations are calculated. Best values for each cluster are selected. After that, the probabilities for different *sequences* are generated. So, the ranking of the identified commands can change based on the whole sequence of commands.

Strategy Analysis: Based on the identified commands, the strategy of the user is being analysed: Different sub-goals are defined in an attack-tree by multiple steps. E.g., the sub-goal *root privileges* can be achieved by exploitation, misconfigured programs, etc. Series of logically related but not necessarily complete intrusion steps are being searched. If a number of subgoals can lead to an intrusion attempt, an alarm is raised.

User Identification: Users of an encrypted connection are identified based on their *keystroke dynamics* recovered from the encrypted network packets.

Policy Conformation: Based on the used sources, commands and the identified user, the accounting and allowed resource usage is verified.

3 Results and Further Work

The modules *Command Evaluation* and *User Identification* had been implemented in a first prototype. Our experiments have shown that both command evaluation and user identification are possible with our proposed method. For the command evaluation, only a limited set is implemented at the moment. This will be advanced especially to the system- and therefore attack-relevant commands. For the strategy analysis, multiple attack-trees will be defined and integrated. After that, a summarizing evaluation will be implemented. The completed prototype will be put into a broad test in the data center of the University.

References

1. Lippmann, R., Webster, S., Stetson, D.: The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection. In: Wespi, A., Vigna, G., Deri, L. (eds.) RAID 2002. LNCS, vol. 2516, p. 307. Springer, Heidelberg (2002)
2. Goh, V.T., Zimmermann, J., Looi, M.: Experimenting with an Intrusion Detection System for Encrypted Networks. Int. J. Business Intelligence and Data Mining 5(2), 172–191 (2010)

Towards Automatic Deduction and Event Reconstruction Using Forensic Lucid and Probabilities to Encode the IDS Evidence

Serguei A. Mokhov, Joey Paquet, and Mourad Debbabi

Concordia University, Montréal, Québec, Canada,
{mokhov,paquet,debbabi}@encs.concordia.ca

Introduction. We apply the theoretical framework and formal model of the observation tuple with the credibility weight for forensic analysis of the IDS data and the corresponding event reconstruction. Forensic Lucid – a forensic case modeling and specification language is used for the task. In the ongoing theoretical and practical work, Forensic Lucid is augmented with the Dempster-Shafer theory of mathematical evidence to include the credibility factors of the evidential IDS observations. Forensic Lucid's toolset is practically being implemented within the General Intensional Programming System (GIPSY) and the probabilistic model-checking tool PRISM as a backend to compile the Forensic Lucid model into the PRISM's code and model-check it. This work may also help with further generalization of the testing methodology of IDSs [10].

Overview. Encoding and modeling large volumes of network and other data related to intrusion detection with Forensic Lucid for the purpose of event correlation and reconstruction along with trustworthiness factors (e.g. the likelihood of logs being altered by an intruder) in a common specification of the evidential statement context and a digital crime scene is an important step in the incident analysis and response. One goal is to able to collect the intrusion-related evidence as the Forensic Lucid's evidential statement from diverse sources like Snort, netflows, pcap's data, etc. to do the follow up investigation and event reconstruction. Another goal is to either be interactive with an investigator present, or fully automated in an autonomous IDS with self-forensics [9].

Background. In the first formal approach about automated cyberforensic case reasoning and event reconstruction, Gladyshev et al. created a finite-state automata (FSA) model [3] to encode the evidence and witness accounts of an incident in order to combine them into an *evidential statement*. Then, they modeled the FSA of a particular case, and, verified if certain claim agrees with the evidential statement, and if it does, list possible event sequences that explain that claim [3]. This was followed by the formal log analysis approach by Arasteh et al [1]. Another earlier work suggested a mathematical theory of evidence by Dempster, Shafer and others [4,12], where factors like credibility play a role in the evaluation, which Gladyshev lacked. Thirdly, another earlier work on intensional logics and programming provided a formal model that throughout its evolution placed the context as a first-class value in language expressions in the system, called Lucid that has produced various Lucid dialects and context-aware systems, such as GIPSY [2,13,11]. Thus, we blended the three together – we augmented the Gladyshev's formalization with the credibility weights and we encode the IDS evidence as a higher-order context (HOC) in the Forensic Lucid language. We then translate a

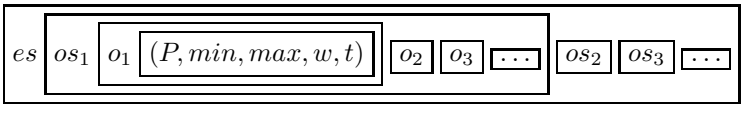
S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 508–509, 2010.

Forensic Lucid specification into the PRISM specification, which is a probabilistic automata evaluation and model-checking system and building a PoC expert system bound to it in CLIPS. Some own work done includes [7,5,8,9].

Computing credibility weights. The notion of an observation is formalized in Equation 1 where w is the credibility weight of that observation, and t is an optional wall-clock timestamp. With $w = 1$ the o would be equivalent to the original model proposed by Gladyshev. We then define the total credibility of an observation sequence as an average of all the weights in this observation sequence. The IDS evidence with higher scores of W have higher credibility.

$$o = (P, \min, \max, w, t) \quad (1) \qquad W_{naive} = \frac{\sum(w_i)}{n} \quad (2)$$

Higher-order context. HOCs represent nested contexts, e.g. as shown in Equation 3 by modeling the evidential statement es containing observation sequences os containing observations o for forensic specification evaluation. In Forensic Lucid it is expressed following the traditional Lucid syntax with modifications adapted from MARFL [6].



(3)

References

1. Arasteh, A.R., Debbabi, M., Sakha, A., Saleh, M.: Analyzing multiple logs for forensic evidence. Digital Investigation Journal 4(1), 82–91 (2007)
2. Ashcroft, E.A., Faustini, A., Jagannathan, R., Wadge, W.W.: Multidimensional, Declarative Programming. Oxford University Press, London (1995)
3. Gladyshev, P., Patel, A.: Finite state machine approach to digital event reconstruction. Digital Investigation Journal 2(1) (2004)
4. Haenni, R., Kohlas, J., Lehmann, N.: Probabilistic argumentation systems. Tech. rep., Institute of Informatics, University of Fribourg, Fribourg, Switzerland (October 1999)
5. Mokhov, S.A.: Encoding forensic multimedia evidence from MARF applications as Forensic Lucid expressions. In: CISSE 2008, pp. 413–416. Springer, Heidelberg (December 2008)
6. Mokhov, S.A.: Towards syntax and semantics of hierarchical contexts in multimedia processing applications using MARFL. In: COMPSAC, pp. 1288–1294. IEEE CS, Los Alamitos (2008)
7. Mokhov, S.A., Paquet, J., Debbabi, M.: Formally specifying operational semantics and language constructs of Forensic Lucid. In: IMF 2008, pp. 197–216. GI (September 2008)
8. Mokhov, S.A., Paquet, J., Debbabi, M.: Reasoning about a simulated printer case investigation with Forensic Lucid. In: HSC 2009. SCS (October 2009) (to appear)
9. Mokhov, S.A., Vassev, E.: Self-forensics through case studies of small to medium software systems. In: IMF 2009, pp. 128–141. IEEE CS, Los Alamitos (2009)
10. Otrok, H., Paquet, J., Debbabi, M., Bhattacharya, P.: Testing intrusion detection systems in MANET: A comprehensive study. In: CNSR 2007, pp. 364–371. IEEE CS, Los Alamitos (2007)
11. Paquet, J., Mokhov, S.A., Tong, X.: Design and implementation of context calculus in the GIPSY environment. In: COMPSAC 2008, pp. 1278–1283. IEEE CS, Los Alamitos (2008)
12. Shafer, G.: The Mathematical Theory of Evidence. Princeton University Press, Princeton (1976)
13. Wan, K.: Lucx: Lucid Enriched with Context. Ph.D. thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada (2006)

Toward Specification-Based Intrusion Detection for Web Applications

Salman Niksefat, Mohammad Mahdi Ahaniha, Babak Sadeghiyan, and Mehdi Shajari

Amirkabir University of Technology
{niksefat,mm.ahaniha,basadegh,mshajari}@aut.ac.ir

1 Introduction

In specification-based detection the correct behavior of a system is modeled formally and would be later verified during system operation for detecting anomalies. In this paper we argue that comparing to anomaly and signature-based approaches, specification-based approach is an appropriate and precise way to build IDSes for web applications. This is due to standardized nature of web architecture including protocols (HTTP, SOAP) and data formats (HTML, XHTML, XML), which makes the challenging task of formal specification feasible. In this paper we propose a novel architecture based on ICAP protocol for a specification-based web application IDS, in which input parameters as well as the output content of a web application are specified formally by regular expressions and the IDS verifies the specification when users have interactions with the application.

A more precise and comprehensive specification makes the IDS engine more powerful and increase the detection rate while decrease the false alarms. A correct specification that exactly matches the real behavior of the system is very important. If the specification is so strict then some normal behavior of the system may be detected as malicious activity and false positives arise. On the other hand, If the specification is so loose or general, then some abnormal behavior of the system may be considered as normal activity and it causes false negatives. Because of the variety of systems and normal behaviors, designing a general specification-based IDS with formal specifications of all normal activities is generally so complicated and imprecise. So researchers mainly focus on a specific system or network protocol and try to formalize the specifications in order to build a specification-based IDS[1].

2 Formal Specification of Web Applications

The standardized nature of web application protocols and data formats makes the challenging work of specification feasible. For building a specification-based IDS for a web application we propose to create the formal specification in the following areas:

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 510–511, 2010.

– **Input Parameters and Values:** Each web application has a number of input parameters. These input parameters and their associated valid values can be identified from design or implementation documents or can be possibly extracted by code analysis tool. To formally specify the input parameters we can utilize various computation models used in computability theory such as regular expressions, finite state machines or push-down automata.
– **Output Content:** By formal specification of the output content and enforcing this specification on our IDS it is possible to detect and prevent attacks such as cross-site scripting (XSS), SQL Injection and information leakages(directory traversal, error pages, etc). Similar to specification of input parameters, the output content can be specified using various computation models.

3 Proposed Architecture

Our idea for building a specification-based IDS is using the Internet Content Adaptation Protocol (ICAP) as well as a middle proxy system such as Squid to deliver the requests and responses to the IDS analysis engine. This idea maximize the interoperability and minimize the implementation overhead of our proposed architecture. This architecture allows the detection ans also prevention of attacks on web applications (Fig.1). When a web client sends a request, the middle proxy machine receives this request, encapsulates it in an ICAP request and sends it the IDS analysis engine. The IDS analysis engine verifies the correctness of the request and either rejects it or forward it to the target web server. The correctness of the responses is verified in the same way.

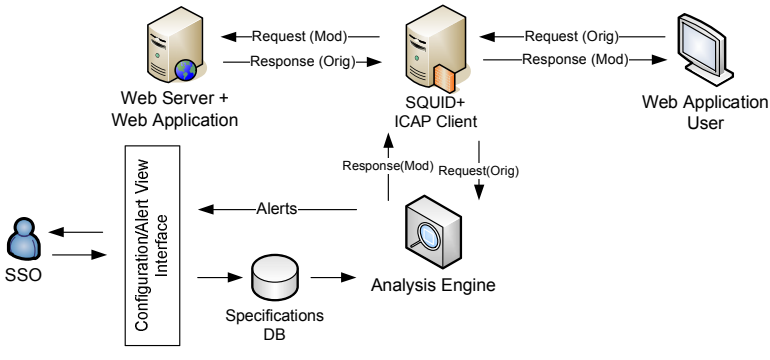


Fig. 1. Proposed Architecture

Reference

1. Orset, J., Alcalde, B., Cavalli, A.: An EFSM-based intrusion detection system for ad hoc networks. In: Automated Technology for Verification and Analysis, pp. 400–413 (2005)

Toward Whole-System Dynamic Analysis for ARM-Based Mobile Devices

Ryan Whelan and David Kaeli

Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, USA
rwhelan@coe.neu.edu, kaeli@ece.neu.edu

Abstract. The ARM architecture is presently the chipset of choice for today's smartphones - this demand has spurred new advances in functionality and services, and as the number of smartphones increases, so has the number of applications being migrated to them. As a result, the amount of malware targeting them will also increase. We present our preliminary work on an ARM-based dynamic profiling platform that allows analysts to study malware that targets ARM-based smartphone systems.

Mobile malware exploits have only begun to appear recently, but analysts expect this trend to accelerate in future years as smartphones begin to dominate the mobile communications market. Smartphones introduce additional attack vectors unavailable on PCs, including Short Messaging Service (SMS), Multimedia Messaging Service (MMS), and Bluetooth. While security experts (especially white hat hackers) have begun to exploit and disclose these vulnerabilities so they can be patched, a new group of embedded systems black hat hackers will soon emerge given the personal and financial data being managed from these systems. As new mobile malware culture continues to mature, security analysts will need a platform that will allow them to study exploits and intrusions. At the moment, a complete ARM-based security analysis platform does not exist that is appropriate for studying mobile malware.

In this paper we report on our implementation of such a platform that is based on QEMU [1]. QEMU provides a *whole-system* emulator for many guest architectures, including ARM. Since the open source development emulator for the Android smartphone operating system is based on QEMU with an ARM guest, we have chosen this environment to develop deep introspection and analysis capabilities for Android. The design of our environment leverages TEMU, an open source dynamic analysis platform based on QEMU for x86 guests [3]. Using QEMU for instrumentation is ideal since it uses dynamic binary translation, which translates execution on the guest architecture to the host architecture at runtime (e.g., ARM on x86). This level of implementation granularity presents an opportunity for fine-grained instrumentation, profiling, and information flow tracking where custom analysis code can be executed along with each guest instruction. Dynamic information flow tracking (i.e., taint tracking) can provide insight into specific events on a system given that data in memory, disk, and

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 512–513, 2010.

registers are instrumented. Any input to the system can then be tracked according to the implemented policy. Given the fact that Android is one of the most popular operating systems on smartphones today, we anticipate that our fully instrumented Android system will be adopted by the mobile security community to study new classes of malware and assist with making tomorrow's smartphones more secure.

On our platform, we have implemented the necessary extensions to dynamically inspect the internal state of an Android guest, and we have begun to evaluate a number of potential threats on ARM-based mobile devices such as alphanumeric ARM shellcode [5], and a kernel module (similar to a rootkit) that hides processes [4]. With our trusted view into the system, we can identify the shell spawned by the shellcode, list processes we've hidden, and generate a rich instruction trace. We obtain our trusted view into the system by analyzing the memory image of the guest and reconstructing the relevant kernel data structures. Our current focus is to address SMS workloads as a portal for additional attack vectors. Recent work has shown that certain SMS messages can render phones inoperable [2], and that worm propagation over Bluetooth is a serious problem that needs to be addressed.

Since the Android emulator provides a mechanism to send SMS messages to the guest, we are approaching the SMS problem by implementing an SMS fuzzing utility, along with a tainting scheme that keeps track of the SMS data propagation through the system. Our preliminary SMS fuzzing can repeatedly and reliably crash the Android process that handles SMS and MMS messages. Once our tainting scheme is fully implemented, it will mark all data derived from SMS input as untrusted and carefully inspect the guest for execution of instructions possessing tainted operands. We will then be provided with a rich profile that will allow the analyst to identify the root cause of this attack (and the associated software bug or vulnerability). We feel it is critical to have the ability to carefully inspect mobile malware *before* it becomes widespread and disables large segments in this market. Our implementation is the first whole-system platform that allows for dynamic analysis of malware and the potential for discovery of new vulnerabilities on popular mobile devices.

References

1. Bellard, F.: Qemu, a fast and portable dynamic translator. In: USENIX 2005 (April 2005)
2. Mulliner, C., Miller, C.: Fuzzing the phone in your phone. In: Black Hat (June 2009)
3. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security, Hyderabad, India (December 2008)
4. ubra: Process hiding and the linux scheduler. In: Phrack, vol. 63 (January 2005)
5. Younan, Y., Philippaerts, P.: Alphanumeric risc arm shellcode. In: Phrack, vol. 66 (November 2009)

Using IRP for Malware Detection

FuYong Zhang, DeYu Qi, and JingLin Hu

Research Institute of Computer Systems at South China University of Technology,
510640 GuangZhou, GuangDong, China
{z.fuyong,qideyu,h.jinglin}@mail.scut.edu.cn

Abstract. Run-time malware detection strategies are efficient and robust, which get more and more attention. In this paper, we use I/O Request Package (IRP) sequences for malware detection. N-gram will be used to analyze IRP sequences for feature extraction. Integrated use of Negative Selection Algorithm (NSA) and Positive Selection Algorithm (PSA), we get more than 96% true positive rate and 0% false positive rate, by a selection of n-gram sequences which only exist in malware IRP sequences.

1 Introduction

The rapid increase in the number of malware has made manual methods of disassembly or reverse engineering can't afford. So security experts focus on the efficient and robust run-time malware detection strategies, by analyzing API calls of real malware and benign processes running on operating system. However, some researchers [2] use the API call capture tool run in user mode, which only can capture API calls in user mode, but not work with API calls in kernel mode. So they can't detect malware which run in kernel mode. Regardless of the program to run in user mode or kernel mode, as long as it exists I/O request will generate the IRP, so we can analyze IRP sequences to distinguish malware and benign.

2 Our Method

We developed an IRP capture tool MBMAS [1] based on kernel driver technology. It can capture processes information created by running programs and the IRPs of each running processes. The statistical analysis of IRPs reveals a total of 30 different types of IRP.

We use 4-gram as detector. As long as there is a sequence has the 4-gram as subsequence, they are match. In the beginning, all permutation of 4-grams are generated as candidate detectors. The first method is, using only NSA to filter out detectors which match self, the rest are mature detectors. The second method is, first using NSA to filter out detectors which match self, then using PSA to select detectors which match nonself. The final detectors are only exist in nonself sequences. Figure 1 is the statistics of unique 4-gram sequences with the total number of IRP growing.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 514–515, 2010.

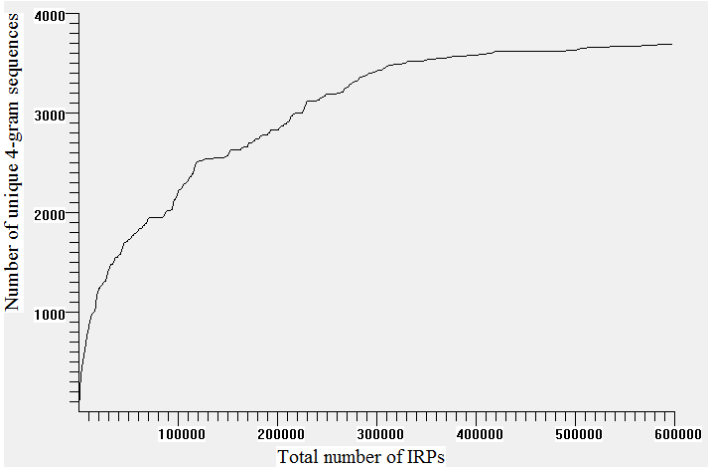


Fig. 1. Unique 4-gram sequences with the total number of IRP growing

3 Experiment

We have collected 600 malware and 300 benign Windows executables. 300 malware come from VX Heaven [3], another 300 are collected from Internet. All 900 files are divided into 2 groups, grpup1 has 200 benign, 200 malware from VX Heaven and 200 malware from Internet, and group2 has 100 benign, 100 malware from VX Heaven and 100 malware from Internet. Group1 will serve as training data and group2 will serve as testing data. In the test, we divide group2 into two groups, benign + VX Heaven malware and benign + Internet malware.

We use 200 benign files in group1 to do self-tolerance and get 807368 mature detectors. Using group2 as testing data, we get 96% true positive rate for benign + VX Heaven malware, 99% true positive rate for benign + Internet malware, and 9% false positive rate. Using the 807368 detectors as candidate detectors, we selected 311 detectors which match at least one of 400 malware in group1 by PSA. Using group2 as testing data, we get exactly the same true positive rate as before, and the false positive rate is 0%.

References

1. Zhang, F.Y., Qi, D.Y., Hu, J.L.: MBMAS: A System for Malware Behavior Monitor and Analysis. In: International Symposium on Computer Network and Multimedia Technology (CNMT 2009), pp. 1–4 (2009)
2. Manzoor, S., Shafiq, M.Z., Tabish, S.M., Farooq, M.: A sense of 'danger' for windows processes. In: Andrews, P.S., Timmis, J., Owens, N.D.L., Aickelin, U., Hart, E., Hone, A., Tyrrell, A.M. (eds.) Artificial Immune Systems. LNAI, LNBI, vol. 5666, pp. 220–233. Springer, Heidelberg (2009)
3. VX Heaven, `http://vx.netlux.org`

Author Index

Ahaniha, Mohammad Mahdi 510
Ahmed, Ezzat 486
Aiken, Alex 360
Ali, Sardar 1
AlShaikh, Raed 486
Antonakakis, Manos 18
Asplund, Mikael 339

Bailey, Michael 138
Balduzzi, Marco 422
Balzarotti, Davide 422
Bellmor, Justin 18
Bertino, Elisa 402
Bolzoni, Damiano 500
Bowen, Brian M. 118
Braje, Timothy 218

Callegari, Christian 490
Cavallaro, Lorenzo 297
Čeleda, Pavel 488
Chen, Kuan-Ta 464
Chen, Shenlong 484
Chen, Weifeng 502
Connelly, Christopher 218
Cova, Marco 442
Crispo, Bruno 198
Cucurull, Jordi 339
Cunningham, Robert K. 218

Dacier, Marc 442
Dagon, David 18
Debbabi, Mourad 508
Ding, Liping 502
Drašar, Martin 488
Dreo Rodosek, Gabi 505

Fattori, Aristide 297

Ganapathy, Vinod 58
Gao, Debin 238
Gazzarrini, Loris 490
Ghosh, Anup 158
Giffin, Jonathon 97
Giordano, Stefano 490
Giuffrida, Cristiano 198
Giura, Paul 277
Guo, Li 498

Hadžiosmanović, Dina 500
Haq, Irfan Ul 1
Hartel, Pieter 500
Holz, Thorsten 422
Hsu, Ching-Hsiang 464
Huang, Chun-Ying 464
Hu, JingLin 514

Ioannidis, Sotiris 79

Jahanian, Farnam 138
Jiang, Jianchun 502
Jiang, Xuxian 178

Kaeli, David 512
Kamra, Ashish 402
Karim, Rezwana 58
Kemerlis, Vasileios P. 118
Keromytis, Angelos D. 118, 442
Khan, Hassan 1
Khayam, Syed Ali 1
Killourhy, Kevin 256
Kirda, Engin 422
Koch, Robert 505
Kong, Ying 494
Krejčí, Radek 488
Kruegel, Christopher 422
Kulkarni, Ashutosh V. 360

Lee, Wenke 18
Leita, Corrado 442
Li, Jun 38
Li, Peng 238
Liu, Limin 238
Liu, Qixu 494
Luo, Xiapu 18

Ma, Qiang 492
Martignoni, Lorenze 297
Mathew, Sunu 382
Matsuura, Kanta 496

Maxion, Roy 256
Memon, Nasir 277
Miller, Barton P. 317
Mokhov, Serguei A. 508

Nadjm-Tehrani, Simin 339
Ngo, Hung Q. 382
Niksefat, Salman 510

Oliner, Adam J. 360
Ortolani, Stefano 198

Pagano, Michele 490
Paleari, Roberto 297
Paquet, Joey 508
Pepe, Teresa 490
Perdisci, Roberto 18
Petropoulos, Michalis 382
Platzer, Christian 422
Prabhu, Pratap 118

Qi, DeYu 514

Rabek, Jesse C. 218
Reiter, Michael K. 238
Rhee, Junghwan 178
Riley, Ryan 178
Rossey, Lee M. 218
Roundy, Kevin A. 317

Sadeghiyan, Babak 510
Shajari, Mehdi 510
Sidiroglou, Stelios 118
Smith, Randy 58
Sqalli, Mohammed 486
Srivastava, Abhinav 97
Stafford, Shad 38
Stavrou, Angelos 158
Stolfo, Salvatore J. 118

Takurou, Hosoi 496
Thonnard, Olivier 442

Upadhyaya, Shambhu 382

Vander Weele, Eric 138
Vasiliadis, Giorgos 79
Vykopal, Jan 488

Wang, Jiang 158
Wang, Yipeng 498
Whelan, Ryan 512
Wright, Charles V. 218

Xing, Luyi 484
Xu, Dongyan 178
Xu, Kui 492
Xu, Yunjing 138

Yang, Liu 58
Yao, Danfeng (Daphne) 492

Zhang, FuYong 514
Zhang, Yuqing 484, 494
Zhang, Zhibin 498