Analysis of COTS for Security Vulnerability Remediation

Gogul Balakrishnan Mihai Christodorescu Vinod Ganapathy
Jonathon T. Giffin Shai Rubin Hao Wang
Somesh Jha[*] Barton P. Miller[†] Thomas Reps[‡]

Computer Sciences Department, University of Wisconsin
1210 West Dayton Street, Madison, Wisconsin 53706

Abstract

The increased use of untrusted, externally-developed program code is reshaping our notions of privacy and organizational boundaries. The use of such public domain and *commercial off-the-shelf* (*COTS*) components offers an organization several advantages, such as decreased development time and increased flexibility during implementation. However, their rash deployment poses two critical risks. First, COTS components may contain or enable vulnerabilities that can be successfully exploited by malicious attackers. Second, COTS components may accidentally or deliberately leak sensitive information. Vulnerability analysis and information-flow analysis address these two risks respectively. In the WiSA project at the University of Wisconsin–Madison, we are developing analysis techniques to address these risks.

1 Introduction

The increased use of untrusted, externally-developed program code is reshaping our notion of privacy and organizational boundaries. The use of such public domain and *commercial off-the-shelf* (*COTS*) components has obvious advantages, such as reduced development time. However, COTS components expose an organization to risks. A component should be given enough access to do its job, but no more (this is known as the *principle of least privilege*). Moreover, organizations that deal with sensitive information should protect this information: organizations should create an *information enclave* that enforces privacy policies. In a closed software process, it is possible to enforce the principle of least privilege and the policies of an information enclave through strict code inspection and coding practices. However, COTS components are by definition developed by other groups, and therefore the organization that uses these components has no control over the coding practices of the developers. Therefore, there is a need for analysis tools to scrutinize COTS components to ensure that they do not contain harmful vulnerabilities or leak sensitive information. There are two classes of analysis techniques: vulnerability analysis and information-flow analysis.

The *Wisconsin Safety Analysis (WiSA)* project lead by S. Jha, B. Miller, and T. Reps is developing analysis techniques especially suited for COTS components.[1] First, we identify requirements for the task of analyzing COTS components. These requirements drive our technical approaches. Our techniques are designed to be multi-lingual (capable of handling multiple languages), handle a wide range of security and privacy policies, and balance accuracy and scalability. We achieve these goals by combining techniques from static analysis (such as program slicing, shape analysis, and alias analysis), model checking, specifications for expressing security policies (such as security automata), and formalisms for expressing information flow (such as secure flow typing and decentralized labels). Combinations of these techniques are required to address the challenges posed by vulnerability and information flow analysis of COTS components. These analysis techniques provide a comprehensive analysis of COTS components and thus reduce the risk due to COTS deployment.

[*] jha@cs.wisc.edu
[†] bart@cs.wisc.edu
[‡] reps@cs.wisc.edu
[1] Detailed information about WiSA can be found at http://www.cs.wisc.edu/wisa.

The basic goals of the WiSA project are:

- **Multi-lingual analysis**
  COTS components, by definition, are developed by remote organizations. Therefore, analysis techniques cannot assume that all COTS components will be written in one programming language. Hence, techniques for analyzing COTS components should be multi-lingual, i.e., capable of working with several different languages. We achieve this goal by analyzing binary code directly or by compiling untrusted source code into an intermediate form which is then analyzed.

- **Balance accuracy and scalability**
  The analysis techniques should be able to handle COTS components of realistic size. We have designed techniques that balance accuracy with scalability. Accuracy is achieved by combining several sophisticated program analysis techniques, such as shape analysis, aliasing analysis, and type inferencing. Scalability is addressed by using techniques for handling composition of components and targeting the analysis to any properties of interest.

- **Support a wide range of safety and privacy policies**
  Security requirements of COTS components depend on the context of their use. As part of this project we plan to develop specification languages that can express safety and privacy related policies. Specifications related to safety express *discretionary access control* policies. *Mandatory access control* policies control flows of sensitive information and can also be expressed in our specification language. Hence, an analyst can express a variety of properties in our specification language.

- **Potential for analyzing composition of components**
  Most real systems are composed of several components. We plan to develop a framework so that components can be analyzed separately and then the analysis results can be integrated. For this purpose, we will base our methodology on *rely-guarantee* reasoning. Rely-guarantee reasoning has been used in software engineering and verification of concurrent systems exactly for this purpose. This will also address the scalability issue.

We have made partial progress towards some of these goals. We have developed a static-analysis and rewriting infrastructure for `x86` binaries. This infrastructure can be used for a variety of tasks, such as model-based intrusion detection, testing malware detectors, and semantics-aware malware detection. A short description of this infrastructure appears in the following section.

2 Binary Analysis Infrastructure

A considerable amount of recent research activity [2–4, 7, 8, 11, 12, 17] has developed analysis tools to find bugs and security vulnerabilities in source code. When attempting to apply such analysis techniques to executables, investigators already encounter a challenging program-analysis problem. The model-checking community would consider the problem to be that of *model extraction*: the tools need to extract a suitable model from the executable. From the perspective of the compiler community, the problem is one of *IR recovery*: tools need to recover Intermediate Representations (IR) from the executable that are similar to those that would be available from source code analysis.

Successful analysis of binary code requires new solutions to this problem. The commercial disassembler IDAPro is a start: IDAPro provides an initial estimate of the IR. However, the IR that IDAPro constructs is incomplete in critical ways that limits its ability to serve as a foundation for further analysis.

- Binary programs frequently transfer control via indirect jumps whose targets are not computed until program run time. IDAPro uses heuristics to resolve indirect jumps. Consequently, it may not resolve all indirect jumps correctly, i.e., it may not find all possible targets of an indirect jump and it even occasionally identifies incorrect targets. Therefore, the control-flow graph constructed by IDAPro is frequently incomplete or outright incorrect. Similar problems occur with IDAPro's resolution of indirect calls; therefore, the call graph is also often incomplete or incorrect. Call graphs and control-flow graphs form the basis of further program analysis. Incorrectness in these graphs will propagate through the analysis and produce bad results.

- IDAPro does not provide a safe estimate of what memory locations are used or modified by each instruction in the executable. Such information is important for tools that aid in program understanding or bug finding; its omission limits the success of these tools.

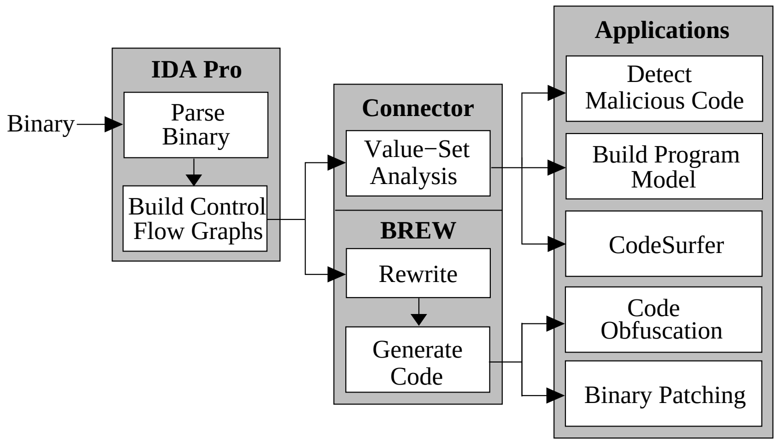

Figure 1: Architecture.

Hence, IDAPro cannot produce a suitable IR for automated program analysis.

We have been developing a static analysis algorithm called *value-set analysis* (VSA) that augments and corrects the information provided by IDAPro in a safe way [1]. Specifically, VSA provides the following information:

- Complete, correct control-flow graphs with indirect jumps resolved safely.
- A call graph with indirect calls resolved safely.
- A set of variable-like entities called *a-locs*.
- Values for pointer variables.
- Used, killed, and possibly-killed variables for nodes in control-flow graphs.

This information is emitted in a format that is suitable for subsequent program analysis applications such as the commercial tool CodeSurfer.

VSA is a flow-sensitive, context-sensitive, abstract-interpretation algorithm parameterized by call-string length [15] that determines a safe over-approximation of the set of numeric values and addresses that memory locations hold at each program point. A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously. This is crucial for analyzing executables because numeric values and addresses are indistinguishable in an executable. VSA has similarities with the pointer-analysis problem that has been studied in great detail for programs written in high-level languages. For each variable v, pointer analysis determines an over-approximation of the set of variables whose addresses v can hold. Similarly, VSA determines an over-approximation of the set of addresses that each a-loc can hold at each program point. On the other hand, VSA also has some of the flavor of numeric static analyses, like constant propagation and interval analysis, where the goal is to over-approximate the integer values that each variable can hold. In addition to information about addresses, VSA determines an over-approximation of the set of integer values that each a-loc can hold at each program point. The result is a safe and correct IR that enables further static program analyses to produce meaningful results for binary programs.

Figure 1 shows how VSA and IDAPro cooperate to produce a base that static analysis application can use to analyze binary programs. IDAPro first processes a binary executable, producing the initial IR that may be incomplete or incorrect. VSA then executes as part of a tool called the *connector*, as it connects IDAPro with static analysis tools in a safe way. The complete, correct IR produced by VSA can then be used by applications performing static program analysis, such as the commercial tool CodeSurfer or research techniques detecting malicious code in executables or building models of expected execution for programs. We will consider these last two applications momentarily.

Alternatively, the IR can be used for binary rewriting, a process that changes the binary code of a program to produce a new executable with altered behavior. Our tool, called *BREW*, alters the IR in the connector and regenerates binary code for the modified program. Binary rewriting applies to tools such as code obfuscators that change program code to increase the difficulty of reverse engineering, and to binary patching tools that repair program bugs without requiring recompilation of source code.

2.1 Applications

Malware detectors, such as virus scanners, identify malicious code hidden within off-the-shelf code and in code shared over communications networks. Despite the importance of malware detectors, there is a dearth of testing techniques to evaluate them. We introduced a technique based on program obfuscation to generate tests for malware detectors [5]. Our technique is geared towards evaluating the resilience of malware detectors to various obfuscation transformations commonly used by hackers to disguise malware. We also demonstrated that a hacker can leverage a malware detector's weakness in handling obfuscation transformations and can extract the signature used by a detector for a specific malware. We evaluated three widely-used commercial virus scanners using our techniques and discovered that the resilience of these scanners to various obfuscations is very poor.

The fundamental deficiency of these commercial virus scanners is their use of pattern-matching approaches to malware detection: these approaches are purely syntactic and ignore the semantics of binary instructions. We developed a malware-detection algorithm that addressed this deficiency by using instruction semantics to detect malicious program traits [6]. Experimental evaluation demonstrated that, with a relatively low run-time overhead, our malware-detection algorithm can detect variants of malware embedded within COTS code. Moreover, our semantics-aware malware detection algorithm is resilient to common obfuscations used by hackers.

Just as a misuse detector identifies attacks contained in program code, a network intrusion detection system (NIDS) detects attacks contained in network traffic. Using network-level obfuscation transformations, we used black-box testing to evaluate the ability of COTS misuse-NIDS products to detect attacks and secure an organization's networks. A misuse-NIDS defines penetration via a table of malicious signatures: if the network traffic matches a signature in the table, an alarm is raised. Both researchers and industry professionals accept that the effectiveness of current off-the-shelf NIDS is questionable. Current NIDS generate many false alarms, and worse (although not always publicized), they miss many real attacks. Our research strives to bring us closer to an effective NIDS: an intrusion detection system that detects the attacks we specify and only those attacks. In our research, we developed formal models and tools that can increase our confidence in NIDS. In the last two years, we have addressed two fundamental problems of NIDS effectiveness: NIDS testing and signature construction.

We formulated a computational model that describes how attackers can generate attack instances that evade a NIDS. Based on this model, we implemented a testing tool that automatically generated new attack instances from known ones [13]. We used this tool to find attack instances that evaded two well-known COTS NIDS: *Snort*, which is a popular NIDS publicly available from SourceFire, and *UnityOne*, which is a commercial NIDS from TippingPoint used by highly secured sites such as the Los Alamos National Lab. In both cases, we exposed vulnerabilities that would have enabled attackers to evade these systems for any TCP-based attack. In response to our findings, both Sourcefire and TippingPoint issued patches to fix their systems.

The signatures that a NIDS uses determine its ability to recognize attacks. We developed a method to systematically construct and evaluate signatures [14]. First, we formally defined the ability of attackers to obfuscate attacks. Then, we combined this formal model with language-based techniques to find loopholes in signatures. To the best of our knowledge, this was the first method that enabled NIDS developers to systematically "debug" the signatures they developed. We showed that, under certain assumptions, the signatures produced are loophole free.

As a complement to network-based intrusion detection, host-based intrusion detection systems identify attempts to exploit program vulnerabilities, frequently by monitoring the program's execution. A model-based or behavioral-based anomaly detector restricts execution to a precomputed model of expected behavior. An execution monitor verifies a stream of system calls generated by the executing program and rejects any call sequences deviating from the model. Constructing a model via static binary program analysis that balances the competing needs of detection ability and efficiency is a challenging task. Non-deterministic finite automaton (NFA) models are efficient to operate, but fail to detect attacks because they do not model the call-return semantics of the program. Pushdown automaton (PDA) models detect more attacks by additionally modeling the program's call stack, but they are inefficient to operate. New models of correct program execution are needed.

We developed a new formal model called the *Dyck model* that preserves the correctness of PDA models but operates with efficiency close to that of NFA models [10, 16]. Our model determinizes previously costly PDA operations modeling the program's call stack [9]. Techniques for determinizing the PDA models essentially incorporate additional program state, such as the program counter and stack activity, into the model. Our results showed that the Dyck model enabled construction of precise program models with performance suitable for online security monitoring. These results vindicated context sensitive models, showing that reasonable efficiency needs not be sacrificed for model precision.

3 Conclusions

We have made good progress towards some of our goals. However, there are some important tasks that should be addressed in the future. We want to improve the robustness and enhance the capabilities of our static-analysis and rewriting infrastructure. Information-flow analysis also remains an important goal which we have not addressed. Compositional analysis of COTS also remains an important goal.

References

[1] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *13th International Conference on Compiler Construction (CC)*, Barcelona, Spain, Apr. 2004.

[2] T. Ball and S. K. Rajamani. The slam toolkit. In *International Conference on Computer Aided Verification (CAV)*, 2001.

[3] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30:775–802, 2000.

[4] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, Nov. 2002.

[5] M. Christodorescu and S. Jha. Testing malware detectors. In *International Symposium on Software Testing and Analysis (ISSTA)*, Boston, MA, July 2004.

[6] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[7] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI)*, New York, NY, 2002.

[8] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating System Design and Implementation (OSDI)*, 2000.

[9] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[10] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *11th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.

[11] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4), 2000.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL)*, 2002.

[13] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of NIDS attacks. In *20th Annual Computer Security Applications Conference (ACSAC)*, Tuscon, AZ, Dec. 2004.

[14] S. Rubin, S. Jha, and B. P. Miller. Language-based generation and evaluation of NIDS signatures. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[15] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[16] W. von Dyck. Gruppentheoretische studien. *Mathematische Annalen*, 20:1–44, 1882.

[17] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, Feb. 2000.