Attacks on Emerging Architectures

by

Steven Craig Hanna Jr.

A dissertation submitted in partial satisfaction

of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Dawn Song, Chair
Professor David Wagner
Professor Brian Carver

Fall 2012

The dissertation of Steven Craig Hanna Jr., titled Attacks on Emerging Architectures, is approved:

Chair ______________________________ Date _____________

______________________________ Date _____________

______________________________ Date _____________

University of California, Berkeley

Attacks on Emerging Architectures

by

Steven Craig Hanna Jr.

Abstract

Attacks on Emerging Architectures

by

Steven Craig Hanna Jr.
Doctor of Philosophy in Computer Science

University of California, Berkeley
Professor Dawn Song, Chair

New architectures require careful examination of security properties in order to assess and contain new threats. In light of this, emerging technologies, such as web APIs, medical devices, and applications on mobile phones, are a new security landscape that has recurring security problems. We develop new techniques to analyze these applications for security vulnerabilities, utilizing techniques including: dynamic symbolic execution, binary analysis and reverse engineering, and wide scale application comparison and classification. We develop *Kudzu*, a system for symbolic execution of JavaScript, and use it to evaluate a wide variety of JavaScript applications in order to find client-side validation vulnerabilities. Secondly, we use this system to evaluate the security, in practice, of new HTML5 primitives. Then, we conduct the first publicly available reverse engineering and security evaluation of a ubiquitous medical device, namely an Automated External Defibrillator. We discovered a wide array of vulnerabilities and we confirm our findings using COTS software components. We offer considerations to help guide future development of medical devices. Finally, we developed *Juxtapp*, a scalable, efficient system for detecting code reuse in Android Applications. Using *Juxtapp* we detected instances of piracy, malware and buggy code reuse among Android applications. We demonstrate that these techniques are useful at discovering and/or preventing attacks, in their respective application domains.

Contents

Acknowledgements

I’d like to thank Jeanine Hanna, Steve Hanna Sr., and Ryan Hanna first and foremost. Without their continued emotional support none of this would have been possible. I’d like to thank Megan O’Brien, for dealing with my whims throughout grad school and constantly supporting my manic behavior, through the best and worst of times. Maggie Brown for generally being awesome, making me happy, and tricking me into working on this dissertation. Mike Perry, Frank Stratton, Chris Grier and Kurt Thomas have been the best hacker/friends/researchers I could imagine, and they’ve been in it for the long haul. Without SIGMil, I likely wouldn’t be here, as it made hacking into something more: research. Adrienne Felt and Prateek Saxena are some of the best people I have ever had the opportunity to work with and I consider myself truly lucky to have colleagues who are so passionate about their work. Also, as a friend, Adrienne’s humor and incredulous glances have driven many revisions. I thank Mark Murphy for telling me to chill out, convincing me out of my impostor syndrome, and giving me a boost when I needed it. I’d also like to thank all of my Berkeley colleagues and innumerable friends who made graduate school far more interesting, loving, exciting, unpredictable, and more adventurous than I ever thought possible. My time here has truly been transformative. In terms of leadership, I’d like to thank David Wagner for the much needed encouragement and support and my adviser Dawn Song, who taught me how to be a scientist and instilled upon me the ability to create good research independently and confidently. Finally, to everyone who has ever challenged me, made me ponderous, knocked me down, brought me back up, educated me, or made an imprint on my being, whether positive or negative, you have helped constantly remind myself of my mantra that has kept me going through all of this: *If you cannot be passionate, why be?*

Chapter 1

Introduction

1.1 Overview

New architectures require careful examination of security properties in order to assess and contain new threats. In light of this, emerging technologies, such as web APIs, medical devices, and applications on mobile phones, are a new security landscape that has recurring security problems. We develop new techniques to analyze these applications for security vulnerabilities, utilizing techniques including: dynamic symbolic execution, binary analysis and reverse engineering, and wide scale application comparison and classification. We demonstrate that these techniques are useful at discovering and/or preventing attacks, in their respective application domains.

First, we discuss *Kudzu* a system for symbolic execution of JavaScript. We detail how we automatically explore client-side JavaScript programs, in order to discover client-side validation vulnerabilities. This new class of vulnerabilities[68] demonstrates how web applications are becoming increasingly complex as they are layered with new APIs and paradigms to developing web applications. We further examine how new HTML5 primitives, those used to create complex client-side JavaScript applications, may be abused by revealing attacks against *postMessage* and the client-side persistent storage API. We offer guiding principles for further primitive development.

Secondly, we discuss the reverse engineering of a publicly available, wide-spread medical device, namely, the CardiacScience G3 Plus Automated External Defibrillator. Our investigation reveals that this new frontier of life-critical devices, specifically the CS G3 Plus, lacked features to make the device secure from attacks. We analyze and reviewed this device for security threats. We use static manual reverse engineering to analyze a medical device for vulnerabilities. We use *BitFuzz*, a trace-based dynamic symbolic fuzzer[2], part of the *BitBlaze* platform, supplied with parameters from static reverse engineering, to confirm our manual analysis. We show that this model of AED is susceptible to many threats and we demonstrate a wide variety of vulnerabilities. These vulnerabilities cause the device to operate outside of a safe spectrum for medical devices. With these vulnerabilities in mind, we offer considerations to guide future designs.

Finally, in the rapidly growing, ubiquitous spectrum of mobile devices, we develop a system *Juxtapp* for discovering similarity among Android applications. The mobile landscape is becoming increasingly dangerous for end users as malicious and pirated software enter third-party application marketplaces. Consequently, a system for detecting exploitable bugs, malware, and pirated software is essential to protect the end user. We develop a scalable system for detecting known bugs, malware, and pirated software. *Juxtapp* is a scalable, efficient system, capable of quickly and accurately detecting similarities among Android applications. We demonstrate its efficacy on a large repository of publicly available applications.

1.1.1 A Symbolic Execution Framework for JavaScript

The web application landscape has rapidly changed with the development of client-side JavaScript applications and new HTML primitives. These rich web applications have a significant fraction of their code written in client-side scripting languages, such as JavaScript. As an increasing fraction of code executes on the client, client-side security vulnerabilities (such as client-side code injection [66, 56, 77, 68]) are becoming a prominent threat. However, a majority of the research on web vulnerabilities so far has focused on server-side application code written in PHP and Java. There is a growing need for powerful analysis tools for the client-side components of web applications. This chapter presents novel techniques and a system for automatically exploring the execution space of client-side JavaScript code. To explore this execution space, our techniques generate new inputs to cover a program's *value space* using dynamic symbolic execution of JavaScript, and to cover its *event space* by automatic GUI exploration.

Dynamic symbolic execution for JavaScript has numerous applications in web security. We focus on one of these applications, automatically finding client-side code injection vulnerabilities. A client-side code injection attack occurs when client-side code passes untrusted input to a dynamic code evaluation construct, without proper validation or sanitization, allowing an attacker to inject JavaScript code that runs with the privileges of a web application.

JavaScript execution space exploration is challenging for many reasons. In particular, JavaScript applications accept many kinds of input, and those inputs are structured just as strings. For instance, a typical application might take user input from form fields, messages from its server via `XMLHttpRequest`, and data from code running concurrently in other browser windows. Each kind of input string has its own format, so developers use a combination of custom routines and third-party libraries to parse and validate the inputs they receive. To effectively explore a program's execution space, a tool must be able to supply values for all of these different kinds of inputs and reason about how they are parsed and validated.

We demonstrate a system capable of automatically exploring both the *value space* and *event space* of JavaScript applications to demonstrate the capability of our system to discover a wide variety of client-side input validation vulnerabilities.

1.1.2 The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives

Furthermore, with the growing demand for interactivity from Web 2.0 applications, web application logic is shifting from the server to the browser. This need to support complex client-side logic and cross-domain interaction has led to a proliferation of new client-side abstractions, such as the proposals in HTML5. A number of major web application providers (including Google and Facebook) have responded by offloading several security-critical parts of their functionality to the client.

However, due to the nascence of these primitives, the security implications of using these new client-side abstractions on the web application's overall security have received little evaluation thus far. To investigate this issue, we selected two primitives as case studies representative of the class of emerging client-side constructs. First, we study systems using *postMessage*, a primitive that enables cross-origin communication within the web browser. Specifically, we analyzed two new purely client-side protocols, namely Google Friend Connect and Facebook Connect, which are layered on *postMessage*. As a second case study, we analyze the usage of client-side storage primitives (such as HTML5 `localStorage`, `webDatabase` API and database storage in Google Gears) by popular applications such as Gmail, Google Docs, Google Buzz and so on.

The *postMessage* API is a message passing mechanism that can be used for secure communication of primitive strings between browser windows. However, if developers do not use the security features of the primitive fully or implicitly trust data arriving on this channel, a variety of attacks can result. We aim to study how consistently this API is used *securely* in practice, by analyzing two prominent client-side protocols using *postMessage*, namely Facebook Connect and Google Friend Connect. To systematically evaluate the security of these protocols, we first reverse engineer the protocol mechanics/semantics as their designs were not documented. In our evaluation, we find that both protocol implementations use the *postMessage* primitive unsafely, opening the protocol to severe confidentiality and integrity attacks. Worse, we observed that several sites using this protocol further widen their attack surface—in one we were able to achieve arbitrary code injection. We were able to concretely demonstrate proof-of-concept exploits that allow unauthorized web sites to compromise users protocol sessions, which can lead to stealing of users data or even injection of arbitrary code into benign web sites using Facebook Connect and Google Friend Connect protocols. In our evaluation, we also observed variations—developers, belonging to the same organization and sometimes of the same application, used the primitives safely in some places while using them unsafely in others. The vulnerabilities in communication primitives have been alluded to in research literature [68, 27]. We responsibly disclose these vulnerabilities and find that in some cases, developers deliberately omit checks in order to ensure compatibility.

As a second representative of a purely client-side abstraction, we study client-side data storage primitives and various applications that rely on these. We find that a large fraction (7 out of 11) of the web applications, including Google Buzz, Gmail and Google Maps, place excessive trust on data in client-side storage. As a result of this reliance, transient attacks (such as a cross-site scripting vulnerability) can persist across sessions (even up to months), while remaining invisible to the web server [48, 76]. In our results, as in the case of the *postMessage* study, we observed a similar inconsistency in developer's sanitization of the dangerous data. Our results show that despite some prior knowledge of the storage vulnerabilities [76], in practice, applications find it difficult to sanitize dangerous data at all places. However, we also note the possibility that in some cases developers may choose to omit sanitizing database input and output because of assurance that other parts of the development framework filter attacks properly.

We observe a common problem with these new client-side primitives: to ensure security, every use of the primitive needs to be accompanied by custom sanity checks. This leads to repeated effort of developing sanity checks by each application that uses the primitive. And, often even within one application similar checks may be distributed throughout the application code, a practice that is prone to errors. We propose the *economy of liabilities* principle in designing security primitives—a primitive must minimize the liability that the user undertakes to ensure application security. For example, in this context, the principle of *economy of liabilities* implies that client-side primitives should internally perform sanitization functionality critical to achieve the intended security property, as much as possible. New primitives today ignore this design principle, achieving security only 'in principle' rather than 'in practice'[1]. We hope the *economy of liabilities* principle will guide the designs of future primitives.

1.1.3 Take Two Software Updates and See Me in the Morning: The Case for Software Security Evaluations of Medical Devices

Life-critical medical devices increasingly contain significant embedded software responsible for safe and effective patient care. Devices range from life-sustaining implantable pacemakers to life-supporting devices such as drug infusion pumps, insulin pumps, and cardiac defibrillator monitors. While recent research analyzes the wireless security but not the software of one implantable medical device, our work considers the embedded software [46]. Little is known about the state of affairs of the security of embedded medical device software itself. Furthermore, medical devices have the unique property that they must do everything they can to *fail open* in order to ensure that even in the presence of an adverse event a life-critical device continues to operate. Studying the susceptibility of medical devices to malware is increasingly important because (1) software in medical devices is becoming increasingly complex; (2) more and more medical devices are becoming networked, with wireless Internet connectivity; (3) more medical devices are evolving from electro-mechanical to software controlled devices. Over the last three decades in the U.S. marketplace, software has played an increasing role in both the function of medical devices and as cause for medical device recalls[39]. In the early 1980s, approximately 6% of recalls were due to computer software issues [83]. The U.S. Food and Drug Administration reports that software caused approximately 18% of the recalls for "510k-approved" devices between 2005 and 2009 [36, p.80]. Software is currently the third leading cause of recalls—just

[1] giving the "Emperor" a false impression of his shiny new clothes

behind design flaws (23%) and manufacturing flaws (37%), but ahead of labeling flaws (12%). We analyzed records in the FDA database of Medical & Radiation Emitting Device Recalls—finding that between 2002–2010, there were 537 recalls of software-based medical devices affecting as many as 1,527,311 individual devices in the U.S. marketplace.[2]

Furthermore, today, many medical devices benefit from a safety blanket of physical isolation. A medical device isolated from other computing devices is less exposed to malware. However, many medical devices are beginning to include wireless communication and Internet connectivity—exposing devices to a much larger surface of potential threats. Therefore, the best way to control the spread of medical malware is to focus on reducing the susceptibility of medical devices by making software more resistant against intentional attacks.

In order to assess the state of current security threats, we perform the first public software security analysis of a medical device, namely, an automatic external defibrillator. AEDs are responsible for evaluating a patient's heart rhythm, and if required, providing a shock to attempt to restore the patient's heart rhythm to normal. We analyzed the (1) Cardiac Science G3 Plus AED model 9390A, an automatic external defibrillator manufactured in 2005[3], and (2) the Windows-based utilities for updating the AED and configuring the device.

We demonstrate a set of vulnerabilities that makes the device potentially susceptible to malware. The discovered vulnerabilities include a buffer overflow that leads to arbitrary code execution, cryptographic flaws in password protection, and a software update mechanism that accepts counterfeit firmware. These issues highlight the impending threat of malware and in that light, we discuss a hypothetical construction of a worm.

We use these vulnerabilities to motivate potential solutions to defend against malware, as well as open challenges in applying computer security principles to medical devices. For instance, a secure update mechanism has the additional challenge of key management and power constraints, making seemingly simple solutions more nuanced. Our assessment demonstrates real vulnerabilities in medical devices and their software and gives a first glimpse into the viability of malware to expect in software-based medical devices.

1.1.4 Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications

As mobile devices (e.g., smartphones, tablets) gain popularity, software marketplaces have become centralized locations for users to download applications. For the Android operating system, Google hosts the official Android Market while Amazon and many others provide third party markets. The wide range of devices that are Android-compatible combined with the open source nature of the Android operating system and development platform have led to explosive growth of the Android market share. As of August of 2011, Android has grown to a 52% market share[87].

[2] Some devices are affected by multiple recalls. An individual device is counted once for each recall.

[3] The G3 Plus is still the flagship AED offered by the manufacturer.

The rapidly increasing volume of applications, increased demand for diversified functionality, and existence of piracy and malware places large obstacles in the way of a healthy and sustainable Android market.

Vulnerable code reuse. To meet the demand for a variety of applications and functionality, developers must adapt their applications to customer needs and balancing the tasks of debugging, all of which can lead to buggy or vulnerable code. Besides coding mistakes, Android developers often misuse coding idioms in Android, either due to copying and pasting of vulnerable code or lack of developer understanding[37, 32]. Even worse, legitimate applications may copy buggy code and libraries directly from programming forums and unwittingly propagate potentially security critical bugs into their programs. For instance, Google has provided sample code to interface with the License Verification Library and the In-Application Billing APIs, which are responsible for verifying that a user is authorized to execute a program and purchasing virtual items within an application, respectively[10, 9]. Google explicitly warns developers that they need to modify certain parts of the code, because the unmodified template code is subject to certain security vulnerabilities and requires developer intervention in order to ensure security properties.

Malware. With the exploding growth in the number of Android applications, the occurrence of Android malware has also increased. As of August 2011, users are 2.5 times more likely to encounter malware on their mobile devices than only 6 months ago and it is estimated that as many as 1 million users have been exposed to malware[15].

Piracy. Furthermore, the third party Android software marketplaces are home to pirated applications[6]. A common occurrence is for an illegitimate author to repackage and rebrand a paid or popular game or application with additional program functionality in order to generate revenue and even execute malicious code.

The current markets usually rely on two approaches to identify and remove potentially dangerous applications: 1) review-based approach, which requires mostly expert manual review and security examination, and 2) reactive approach, e.g., user policing, reporting, and user ratings as indicators that an application may be misleading in its functionality or misbehaving. Given the existence of hundreds of thousands of applications on the markets, neither approach is scalable and reliable enough to mitigate threats to users. To empower and expedite this process, we need an automated analysis of Android applications in order to pare down large application datasets into a small set of noteworthy candidates for further investigation.

Each of the aforementioned problems appears to be unrelated. However, we observe a common invariant among them, namely, code reuse, which sheds light on the fact that a unified approach in detecting common code (or code similarity) may address all of our goals. Using this observation, we propose to build a fast and scalable infrastructure for detecting code reuse in Android applications which allows for 1) early detection and developer notification of known vulnerable or buggy code, 2) detection of instances of known malware, either in isolation or repackaged with an innocuous program, and 3) detection of pirated applications.

It is a challenging task to develop a system to automatically detect code reuse in Android applications. The system must be able to quickly compare code and detect reuse, and scale to hundreds of thousands applications or more; the system need to be resilient to certain levels of code modification and obfuscation, which are common in Android applications; the system should be able to represent the application being compared in a meaningful, accurate way in order to find the so-called needle-in-a-haystack differences in applications, all the while maintaining low false positive and false negative rates.

As a first step solution, we use k-grams of opcode sequences of compiled applications and feature hashing[54, 50] to efficiently tackle the problem at large-scale. k-grams of opcode sequences have been shown to be resilient to certain types of code modification and can be efficiently extracted from applications. Additionally, feature hashing has been shown to work well in dimensionality reduction and classification. We combine this technique with a variety of domain-specific knowledge in order evaluate code reuse, instances of known malware, and piracy in Android applications. We use k-grams and feature hashing combined in order to have a robust and efficient representation of applications. Using this representation, we have a fast way to compute pairwise similarity between applications to detect code reuse among hundreds of thousand of applications.

In light of this, we propose *Juxtapp*, a scalable architecture for quickly detecting code reuse and similarity in Android applications. We implemented our distributed architecture using Hadoop and ran it on Amazon EC2. It is capable of fast, *incremental* additions to the analysis dataset, meaning it is amenable to frequent updates and additions to the pool of applications. We apply Juxtapp to address three different types of problems: vulnerable code reuse, known malware, and piracy. We evaluate Juxtapp's ability to detect these problems on 58,000 applications, ranging in size from hundreds of kilobytes to tens of megabytes, which were collected from the official Android market and the Anzhi third party market[1]. We find that the system performs and scales well.

Chapter 2

A Symbolic Execution Framework for JavaScript

2.1 Introduction

As AJAX applications gain popularity, client-side JavaScript code is becoming increasingly complex. However, few automated vulnerability analysis tools for JavaScript exist. In this chapter, we describe the first system for exploring the execution space of JavaScript code using symbolic execution. To handle JavaScript code's complex use of string operations, we design a new language of string constraints and implement a solver for it. We build an automatic end-to-end tool, Kudzu, and apply it to the problem of finding client-side code injection vulnerabilities. Kudzu explores both the event space, using the GUI explorer and the value space, using our string solver. In experiments on 18 live web applications, Kudzu automatically discovers 2 previously unknown vulnerabilities and 9 more that were previously found only with a manually-constructed test suite.

Contributions[1].

In summary, this chapter makes the following main contributions:

- We build the first symbolic execution engine for JavaScript, using our constraint solver.
- Combining symbolic execution of JavaScript with automatic GUI exploration and other needed components, we build the first end-to-end automated system for exploration of client-side JavaScript.
- We demonstrate the practical use of our implementation by applying it to automatically discovering 11 client-side code injection vulnerabilities, including two that were previously unknown.

[1] **In this chapter, I was partially responsible for writing the path constraint collector, which was supplemented and finished by Devdatta. Prateek, Devdatta and myself instrumented the browser for trace extraction. I wrote the input feedback mechanism in the browser, and the event space explorer.**

2.2 Problem Statement and Overview

In this chapter we state the problem we focus on, exploring the execution space of JavaScript applications; describe one of its applications, finding client-side code injection vulnerabilities; and give an overview of our approach.

Problem statement. We develop techniques to systematically explore the execution space of JavaScript application code.

JavaScript applications often take many kinds of input. We view the input space of a JavaScript program as split into two categories: the *event space* and the *value space*.

- *Event space*. Rich web applications typically define tens to hundreds of JavaScript event handlers, which may execute in any order as a result of user actions such as clicking buttons or submitting forms. Event handler code may check the state of GUI elements (such as check-boxes or selection lists). The ordering of events and the state of the GUI elements together affects the behavior of the application code.
- *Value space*. The values of inputs supplied to a program also determine its behavior. JavaScript has numerous interfaces through which input is received:
    - *User data*. Form fields, text areas, and so on.
    - *URL and cross-window communication abstractions*. Web principals hosted in other windows or frames can communicate with JavaScript code via inter-frame communication abstractions such as URL fragment identifiers and HTML 5's proposed `postMessage`, or via URL parameters.
    - *HTTP channels*. Client-side JavaScript code can exchange data with its originating web server using `XMLHttpRequest`, HTTP cookies, or additional HTTP `GET` or `POST` requests.

This chapter primarily focuses on techniques to systematically explore the value space using symbolic execution of JavaScript, with the goal of generating inputs that exercise new program paths. However, automatically exploring the event space is also required to achieve good coverage. To demonstrate the efficacy of our techniques in an end-to-end system, we combine symbolic execution of JavaScript for the value space with a GUI exploration technique for the event space. This full system is able to automatically explore the combined input space of client-side web application code.

Application: finding client-side code injection vulnerabilities. Exploring a program's execution space has a number of applications in the security of client-side web applications. In this chapter, we focus specifically on one security application, finding client-side code injection vulnerabilities.

Client-side code injection attacks, which are sometimes referred to as DOM-based XSS, occur when client-side code uses untrusted input data in dynamic code evaluation constructs without sufficient validation. Like reflected or stored XSS attacks, client-side code injection vulnerabilities can be used to inject script code chosen by an attacker, giving the attacker the full privileges of the web application. We call the program input that supplies the data for an attack the *untrusted source*, and the potentially vulnerable code evaluation construct the *critical sink*. Examples of critical sinks include `eval` and HTML creation interfaces like `document.write` and `.innerHTML`.

In our threat model, we treat all URLs and cross-window communication abstractions as untrusted sources, as such inputs may be controlled by an untrusted web principal. In addition, we also treat user data as an untrusted source because we aim to find cases where user data may be interpreted as code. The severity of attacks from user-data on the client-side is often less severe than a remote XSS attack, but developers tend to fix these and Kudzu takes a conservative approach of reporting them. HTTP channels such as `XMLHttpRequest` are currently restricted to communicating with a web server from the same domain as the client application, so we do not treat them as untrusted sources. Developers may wish to treat HTTP channels as untrusted in the future when determining susceptibility to cross-channel scripting attacks [28], or when enhanced abstractions (such as the proposed cross-origin XMLHttpRequest [79]) allow cross-domain HTTP communication directly from JavaScript.

To effectively find XSS vulnerabilities, we require two capabilities: (a) generating directed test cases that explore the execution space of the program, and (b) checking, on a given execution path, whether the program validates all untrusted data sufficiently before using it in a critical sink. Custom validation checks and parsing routines are the norm rather than the exception in JavaScript applications, so our tool must check the behavior of validation rather than simply confirming that it is performed.

In previous work, we developed a tool called FLAX which employs taint-guided fuzzing for finding client-side code injection attacks [68]. However, FLAX relies on an external, manually developed test harness to explore the path space. Kudzu, in contrast, automatically generates a test suite that explores the execution space systematically. Kudzu also uses symbolic reasoning (with its constraint solver) to check if the validation logic employed by the application is sufficient to block malicious inputs — this is a one-step mechanism for directed exploit generation as opposed to multiple rounds of undirected fuzzing employed in FLAX. Static analysis techniques have also been employed for JavaScript [43] to reason about multiple paths, but can suffer from false positives and do not produce test inputs or attack instances. Symbolic analyses and model-checking have been used for server-side code [60, 25]; however, the complexity of path conditions we observe requires more expressive symbolic reasoning than supported by tools for server-side code.

Approach Overview. The value space and event space of a web application are two different components of its input space: code reachable by exploring one part of the input space may not be reachable by exploring the other component alone. For instance, exploring the GUI event space results in discovering new views of the web application, but this does not directly affect the coverage that can be achieved by systematically exploring all the paths in the code implementing each view. Conversely, maximizing path coverage is unlikely to discover functionality of the application that only happens when the user explores a different application view. Therefore, Kudzu employs different techniques to explore each part of the input space independently.

Value space exploration. To systematically explore different execution paths, we develop a component that performs dynamic symbolic execution of JavaScript code, and a new constraint solver that offers the desired expressiveness for automatic symbolic reasoning.

In dynamic symbolic execution, certain inputs are treated as symbolic variables. Dynamic symbolic execution differs from normal execution in that while many variable have their usual (*concrete*) values, like 5 for an integer variable, the values of other variables which depend on symbolic inputs are represented by *symbolic* formulas over the symbolic inputs, like $input_1 + 5$. Whenever any of the operands of a JavaScript operation is symbolic, the operation is simulated by creating a formula for the result of the operation in terms of the formulas for the operands. When a symbolic value propagates to the condition of a branch, Kudzu can use its constraint solver to search for an input to the program that would cause the branch to make the opposite choice.

Event space exploration. As a component of Kudzu we develop a GUI explorer that searches the space of all event sequences using a random exploration strategy. Kudzu's GUI explorer component randomly selects an ordering among the user events registered by the web page, and automatically fires these events using an instrumented version of the web browser. Kudzu also has an input-feedback component that can replay the sequence of GUI events explored in any given run, along with feeding new values generated by the constraint solver to the application's data inputs.

Testing for client-side code injection vulnerabilities. For each input explored, Kudzu determines whether there is a flow of data from an untrusted data source to a critical sink. If it finds one, it seeks to determine whether the program sanitizes and/or validates the input correctly to prevent attackers from injecting dangerous elements into the critical sink. Specifically, it attempts to prove that the validation is insufficient by constructing an attack input. Chapter 2.3.2 combines the results of symbolic execution with a specification for attacks to create a constraint solver query. If the constraint solver finds a solution to the query, it represents an attack that can reach the critical sink and exploit a client-side code injection vulnerability.

2.3 End-to-End System Design

Herein, we describe the various components that work together to make a complete Kudzu-based vulnerability-discovery system work. For reference, the relationships between the components are summarized in Figure 2.1.

2.3.1 System Components

First, we discuss the core components that would be used in any application of Kudzu: the *GUI explorer* that generates input events to explore the event space, the *dynamic symbolic interpreter* that performs symbolic execution of JavaScript, the *path constraint extractor* that builds queries based on the results of symbolic execution, the *constraint solver* that

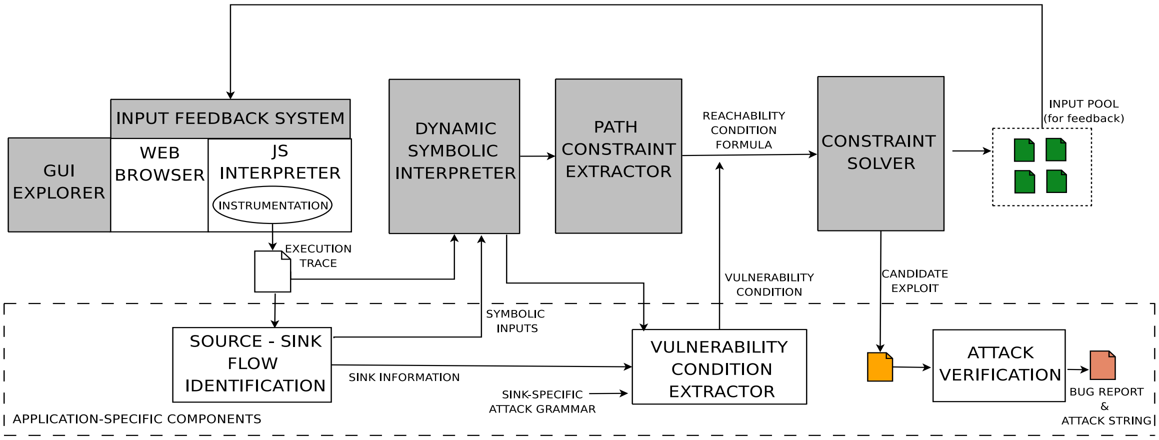

INPUT FEEDBACK SYSTEM
GUI EXPLORER
WEB BROWSER
JS INTERPRETER
INSTRUMENTATION
DYNAMIC SYMBOLIC INTERPRETER
PATH CONSTRAINT EXTRACTOR
REACHABILITY CONDITION FORMULA
CONSTRAINT SOLVER
INPUT POOL (for feedback)
EXECUTION TRACE
SYMBOLIC INPUTS
VULNERABILITY CONDITION
CANDIDATE EXPLOIT
SOURCE - SINK FLOW IDENTIFICATION
SINK INFORMATION
SINK-SPECIFIC ATTACK GRAMMAR
VULNERABILITY CONDITION EXTRACTOR
ATTACK VERIFICATION
BUG REPORT & ATTACK STRING
APPLICATION-SPECIFIC COMPONENTS

Figure 2.1: Architecture diagram for Kudzu. The components drawn in the dashed box perform functions specific to our application of finding client-side code injection. The remaining components are application-agnostic. Components shaded in light gray are the core contribution of this paper.

finds satisfying assignments to those queries, and the *input feedback* component that uses the results from the constraint solver as new program inputs.

The GUI explorer. The first step in automating JavaScript application analysis is exploring the event space of user interactions. Each event corresponds to a user interaction such as clicking a check-box or a button, setting focus on a field, adding data to data fields, clicking a link, and so on. Kudzu currently explores the space of all sequences of events using a random exploration strategy. One of the challenges is to comprehensively detect all events that could result in JavaScript code execution. To address this, Kudzu instruments the browser functions that process HTML elements on the current web page to record when an event handler is created or destroyed. Kudzu's GUI explorer component randomly selects an ordering among the user events registered by the web page and executes them[2]. The random seed can be controlled to replay the same ordering of events. While invoking handlers, the GUI component also generates (benign) random test strings to fill text fields. (Later, symbolic execution will generate new input values for these fields to explore the input space further.) Links that navigate the page away from the application's domain are cancelled, thereby constraining the testing to a single application domain at a time. In the future, we plan to investigate alternative strategies to prioritize the execution of events discovered as well.

Dynamic symbolic interpreter. Kudzu performs dynamic symbolic execution by first recording an execution of the program with concrete inputs, and then symbolically interpreting the recorded execution in a dynamic symbolic interpreter. For recording an execution trace, Kudzu employs an existing instrumentation component [68] implemented in the

[2] Invoking an event handler may invalidate another handler (for instance, when the page navigates as a result). In that case, the invalidated handlers are ignored and if new handlers are created by the event that causes invalidation, these events are explored subsequently.

web browser's JavaScript interpreter. For each JavaScript bytecode instruction executed, it records the semantics of the operation, its operands and operand values in a simplified intermediate language called JASIL[68]. The set of JavaScript operations captured includes all operations on integers, booleans, strings, arrays, as well as control-flow decisions, object types, and calls to browser-native methods. For the second step, dynamic symbolic execution, we have developed from scratch a symbolic interpreter for the recorded JASIL instructions.

Symbolic inputs for Kudzu are configurable to match the needs of an application. For instance, in the application we consider, detecting client-side code injection, all URL data, data received over cross-window communication abstractions, and user data fields are marked symbolic. Symbolic inputs may be strings, integers, or booleans. Symbolic execution proceeds on the JASIL instructions in the order they are recorded in the execution trace. At any point during dynamic symbolic execution, a given operand is either *symbolic* or *concrete*. If the operand is symbolic, it is associated with a *symbolic value*; otherwise, its value is purely concrete and is stored in the dynamic execution trace. When interpreting a JASIL operation in the dynamic symbolic interpreter, the operation is symbolically executed if one or more of its input operands are symbolic. Otherwise the operation of the symbolic interpreter on concrete values would be exactly the same as the real JavaScript interpreter, so we simply reuse the concrete results already stored in the execution trace.

The symbolic value of an operand is a formula that represents its computation from the symbolic inputs. For instance, for the JASIL assignment operation `y := x`, if `x` is symbolic (say, with the value $\mathit{input}_1 + 5$), then symbolic execution of the operation copies this value to `y`, giving `y` the same symbolic value. For an arithmetic operation, say `y := x1 + x2` where `x1` is symbolic (say with value $\mathit{input}_2 + 3$) and `x2` is not (say with the concrete value 7), the symbolic value for `y` is the formula representing the sum $(\mathit{input}_2 + 10)$. Operations over strings and booleans are treated in the same way, generating formulas that involve string operations like `match` and boolean operations like `and`. At this point, string operations are treated simply as uninterpreted functions. During the symbolic execution, whenever the symbolic interpreter encounters an operation outside the supported formula grammar, it forces the destination operand to be concrete. For instance, if the operation is `x = parseFloat(s)` for a symbolic string `s`, `x` and `s` can be replaced with their concrete values from the trace (say, 4.3 and "`4.3`"). This allows symbolic computation to continue for other values in the execution.

Path constraint extractor. The execution trace records each control-flow branch (e.g., `if` statement) encountered during execution, along with the concrete value (true or false) representing whether the branch was taken. During symbolic execution, the path constraint extractor records the corresponding *branch condition* if it is symbolic. As execution continues, the formula formed by conjoining the symbolic branch conditions (negating the conditions of branches that were not taken) is called the *path constraint*. If an input value satisfies the path constraint, then the program execution on that input will follow the same execution path.

To explore a different execution path, Kudzu selects a branch on the execution path and builds a modified path constraint that is the same up to that branch, but that has the negation of that branch condition (later conditions from the original branch are omitted). An input that satisfies this condition will execute along the same path up to the selected branch, and then explore the opposite alternative. There are several strategies for picking the order in which branch conditions can be negated — Kudzu currently uses a generational search strategy [42].

Input feedback. Solving the path constraint formula using the solver creates a new input that explores a new program path. These newly generated inputs must be fed back to the JavaScript program: for instance simulated user inputs must go in their text fields, and GUI events should be replayed in the same sequence as on the original run. The input feedback component is responsible for this task. As a particular HTML element (e.g a text field) in a document is likely allocated a different memory address on every execution, the input feedback component uses XPath [86] and DOM identifiers to uniquely identify HTML elements across executions and feed appropriate values into them. If an input comes from an attribute for a DOM object, the input feedback component sets that attribute when the object is created. If the input comes via a property of an event that is generated by the browser when handling cross-window communication, such as the `origin` and `data` properties of a `postMessage` event, the component updates that property when the JavaScript engine accesses it. Kudzu instruments the web browser to determine the context of accesses, to distinguish between accesses coming from the JavaScript engine and accesses coming from the browser core or instrumentation code.

2.3.2 Application-specific components

Next, we discuss three components that are specialized for the task of finding client-side code injection vulnerabilities: a *sink-source identification* component that determines which critical sinks might receive untrusted input, a *vulnerability condition extractor* that captures domain knowledge about client-side code injection attacks, and the *attack verification* component that checks whether inputs generated by the tool in fact represent exploits.

Sink-source identification. To identify if external inputs are used in critical sink operations such as `eval` or `document.write`, we perform a dynamic data flow analysis on the execution trace. As outlined earlier, we treat all URL data, data received over cross-window communication abstractions (such as `postMessage`), and data filled into user data fields as potentially untrusted. The data flow analysis is similar to a dynamic taint analysis. Any execution trace that reveals a flow of data to a critical sink is subject to further symbolic analysis for exploit generation. We use an existing framework, FLAX, for this instrumentation and taint-tracking [68] in a manner that is faithful to the implementation of JavaScript in the WebKit interpreter.

Vulnerability condition extractor. An input represents an attack against a program if it passes the program's validation checks, but nonetheless implements the attacker's goals (i.e., causes a client-side code injection attack) when it reaches a critical sink. The vulnerability condition extractor collects from the symbolic interpreter a formula representing the (possibly transformed) value used at a critical sink, and combines it with the path constraint to create a formula describing the program's validation of the input.[3]. To determine whether this value constitutes an attack, the vulnerability condition extractor applies a sink-specific vulnerability condition specification, which is a (regular) grammar encoding a set of strings that would constitute an attack against a particular sink. This specification is conjoined with the formula representing the transformed input to create a formula representing values that are still dangerous after the transformation.

For instance, in the case of the `eval` sink, the vulnerability specification asserts that a valid attack should be zero or more statements each terminated by a ';', followed by the payload. Such grammars can be constructed by using publicly available attack patterns [47]. The tool's attack grammars are currently simple and can be extended easily for comprehensiveness and to incorporate new attacks.

To search only for realistic attacks, the specification also incorporates domain knowledge about the possible values of certain inputs. For instance, when a string variable corresponds to the web URL for the application, we assert that the string starts with the same domain as the application.

Attack verification. Kudzu automatically tests the exploit instance by feeding the input back to the application, and checking if the attack payload (such as a script with an alert message) is executed. If this verification fails, Kudzu does not report an alarm.

2.4 Experimental Evaluation

We have built a full implementation of Kudzu using the WebKit browser, with 650, 7430 and 2200 lines of code in the path constraint extraction component, constraint solver, and GUI explorer component, respectively. The system is written in a mixture of C++, Ruby, and OCaml languages.

We evaluate Kudzu with three objectives. One objective is to determine whether Kudzu is practically effective in exploring the execution space of real-world applications and uncovering new code. The second objective is to determine the effectiveness of Kudzu as a stand-alone vulnerability discovery tool — whether Kudzu can automatically find client-side code injection vulnerabilities and prune away false reports. Finally, we measure the efficiency of the constraint solver. Our evaluation results are promising, showing that Kudzu is a powerful system that finds previously unknown vulnerabilities in real-world applications fully automatically.

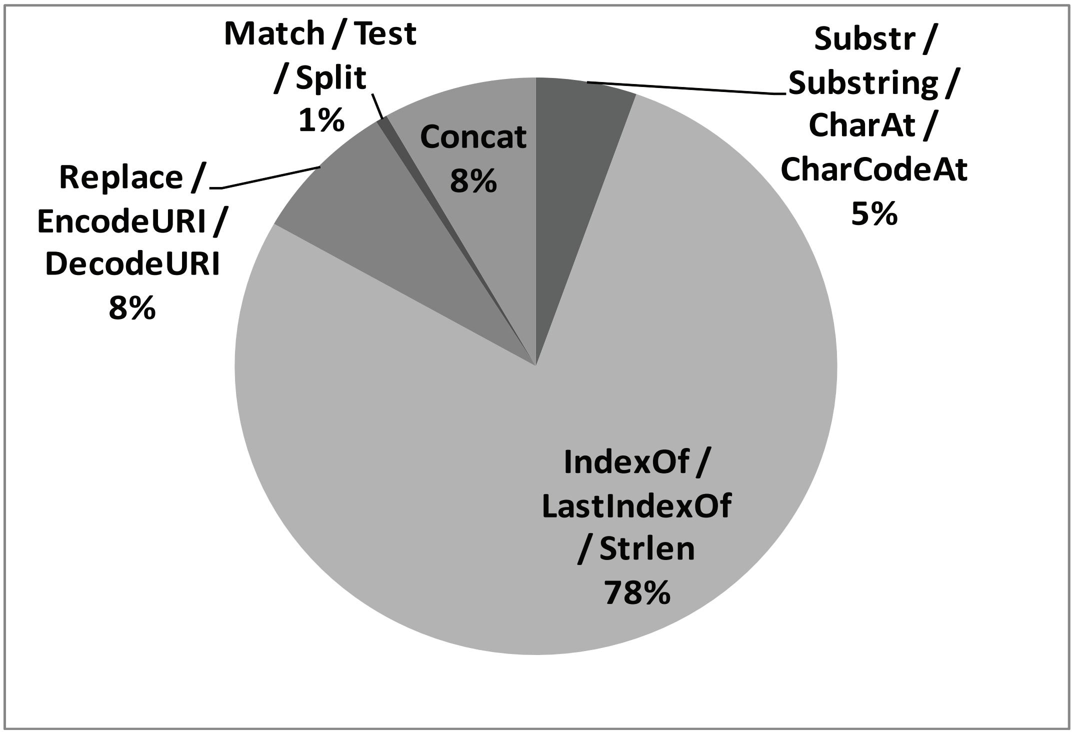


Figure 2.2: Distribution of string operations in our subject applications.

2.4.1 Experiment Setup

We select 18 subject applications consisting of popular iGoogle gadgets and AJAX applications, as these were studied by our previous tool FLAX [68]. FLAX assumes availability of an external (manually developed) test suite to seed its testing; in contrast, Kudzu automatically generates a more comprehensive test suite and finds the points of vulnerability without requiring any external test harness a priori. Further, in our experiments Kudzu discovers 2 new vulnerabilities within a few hours of testing which were missed by the FLAX because of its lack of coverage. In addition, as we show later in this chapter, many of the generated constraints are highly complex and not suitable for manual inspection or fuzzing, whereas Kudzu either asserts the safety of the validation checks or finds exploits for vulnerabilities in one iteration as opposed to many rounds of random testing.

To test each subject application, we seed the system with the URL of the application. For the gadgets, the URLs are the same as those used by iGoogle page to embed the gadget. We configure Kudzu to give a pre-prepared username and login password for applications that required authentication. We report the results for running each application under Kudzu, capping the testing time to a maximum of 6 hours for each application. All tests ran on a Ubuntu 9.10 Linux workstation with 2.2 GHz Intel dual-core processors and 2 GB of RAM.

[3] Sanitization for critical client-side sink operations may happen on the server side (when data is sent back via `XMLHttpRequest`). Our implementation handles this by recognizing such transformations using approximate tainting techniques [68] for data transmitted over `XMLHttpRequest`

Application	# of new inputs	Initial / Final Code Coverage	Bug found
Academia	20	30.27 / 76.47%	✓
AJAXIm	15	49.58 / 77.67%	✓
FaceBook Chat	54	26.85 / 76.84%	-
ParseUri	13	53.90 / 86.10%	✓
Plaxo	31	5.72 / 76.43%	✓
AskAWord	10	29.30 / 67.95 %	✓
Birthday Reminder	27	59.47 / 73.94%	-
Block Notes	457	65.06 / 71.50 %	✓
Calorie Watcher	16	64.54 / 73.53%	-
Expenses Manager	133	61.09 / 76.56%	-
Listy	19	65.31 / 79.73%	✓
Notes LP	25	46.62 / 76.67%	-
Progress Bar	12	63.60 / 75.09%	-
Simple Calculator	1	46.96 / 80.52%	✓
Todo List	15	72.51 / 86.41%	✓
TVGuide	6	30.39 / 75.13%	✓
Word Monkey	20	14.84 / 75.36%	✓
Zip Code Gas	11	59.05 / 74.28%	-
Average	**49**	**46.95 / 76.68%**	**11**

Table 2.1: The top 5 applications are AJAX applications, while the rest are Google/IG gadget applications. Column 2 reports the number of distinct new inputs generated, and column 3 reports the increase in code coverage from the initial run to and the final run.

2.4.2 Results

Table 2.1 presents the final results of testing the subject applications. The summary of our evaluation highlights three features of Kudzu: (a) it automatically discovers new program paths in real applications, significantly enhancing code coverage; (b) it finds 2 client-side code injection in the wild and several in applications that were known to contain vulnerabilities; and (c) Kudzu significantly prunes away most false positives, successfully discarding cases that do employ sufficient validation checks. In our experiments we had no false positives. However, Kudzu can have false positives occur when the symbolic formula in our constraint language does not perfectly match the underlying JavaScript being executed in the browser. This is most prevalent in the transformation of regular expressions to our constraint language, in which we must have explicit and controlled expansions of symbolic formulas, in order to make the solution tractable. We discuss the cause of false positives in Chapter 2.4.2.1.

Characteristics of string operations in our applications.

Constraints arising from our applications have an average of 63 JavaScript string operations, while the remaining are boolean, logical and arithmetic constraints. Figure 2.2

groups the observed string operations by similarity. The largest fraction are operations like `indexOf` that take string inputs and return an integer, which motivate the need for a solver that reasons about integers and strings simultaneously. A significant fraction of the operations, including `substring`, `split` and `replace`, implicitly give rise to new strings from the original one, thereby giving rise to constraints involving multiple string variables. Of the `match`, `split` and `replace` operations, 31% are regular expression based. Over 33% of the regular expressions have one or more capturing parentheses. Capturing parentheses in regular expression based match operations lead to constraints involving multiple string variables, similar to operations such as `split`.

These characteristics show that a significant fraction of the string constraints arising in our target applications require a solver that can reason about multiple string variables. We empirically see examples of complex regular expressions as well as concatenation operations, which stresses the need for our solver that handles both word equations and regular expression constraints. Prior to this work, off-the-shelf solvers did not support word equations and regular expressions simultaneously.

Vulnerability Discovery. Kudzu is able to find client-side code injection vulnerabilities in 11 of the applications tested. 2 of these were not known prior to these experiments and were missed by FLAX. One of them is on a social-networking application (`http://plaxo.com`) that was missed by our FLAX tool because the vulnerability exists on a page linked several clicks away from the initial post-authentication page. The vulnerable code is executed only as part of a feature in which a user sets focus on a text box and uses it to update his or her profile. This is one of the many different ways to update the profile that the application provides. Kudzu found that only one of these ways resulted in a client-side code injection vulnerability, while the rest were safe. In this particular functionality, the application fails to properly validate a string from a `postMessage` event before using it in an `eval` operation. The application implicitly expects to receive this message from a window hosted at a sub-domain of `facebook.com`; however, Kudzu automatically determines that *any* web principal could inject any data string matching the format `FB_msg:.*{.*}`. This subsequently results in code injection because the vulnerable application fails to validate the origin of the sender and the structure of JSON string before its use in `eval`.

The second new vulnerability was found in a ToDo Google/IG gadget. Similar to the previous case, the vulnerability becomes reachable only when a specific value is selected from a dropdown box. This interaction is among many that the gadget provides and we believe that Kudzu's automatic exploration is the key to discovering this use case. In several other cases, such as AjaxIM, the vulnerable code is executed only after several events are executed after initial sign-in page—Kudzu automatically reaches them during its exploration.

Kudzu did not find vulnerabilities in only one case that FLAX reported a bug. This is because the vulnerability was patched in the time period between our experimental evaluation of FLAX and Kudzu.

Code and Event-space Coverage. Table 2.1 shows the code coverage by executing the initial URL, and the final coverage after the test period. Measuring code coverage in a

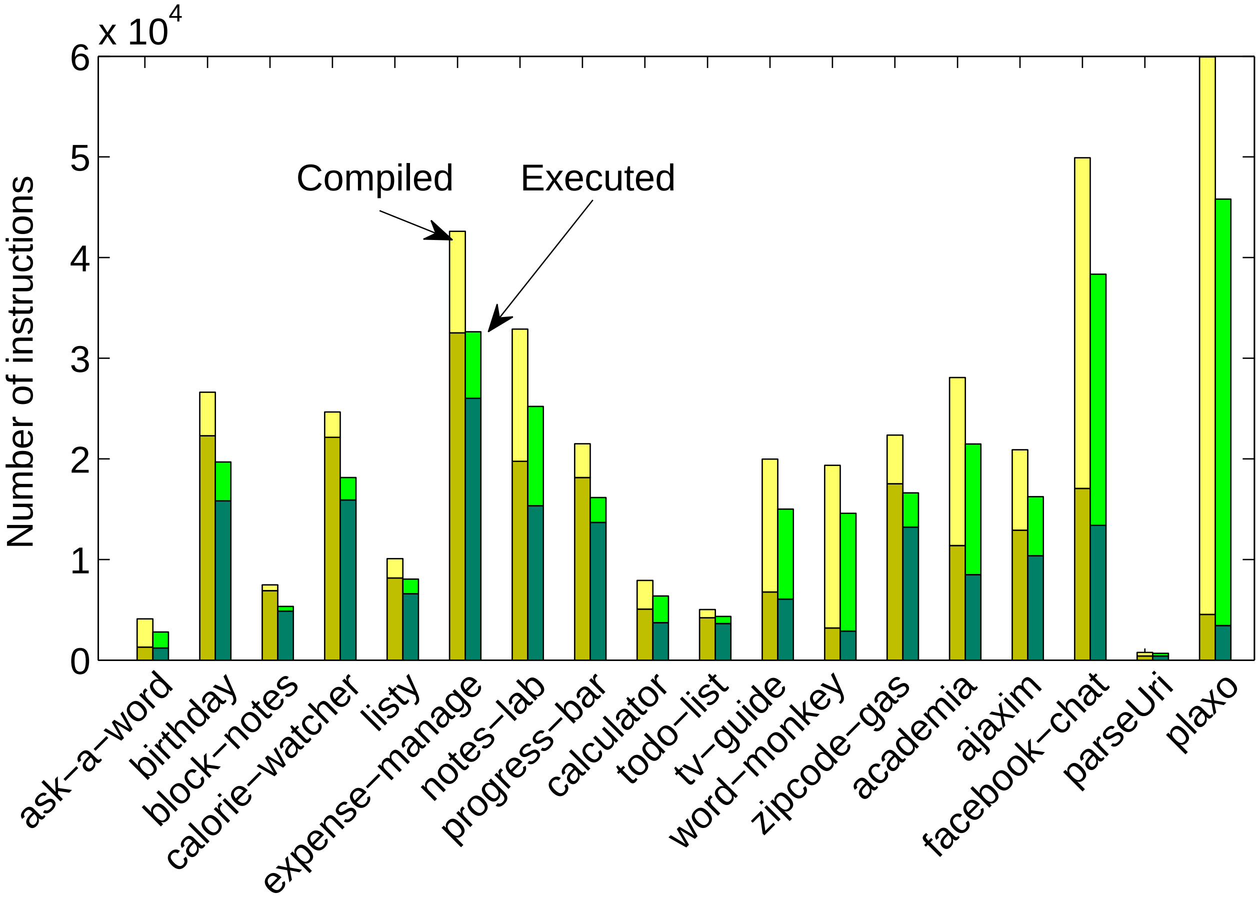


Figure 2.3: For each application, the lower-left and lower-right bars represent the amount of code compiled and executed initially when the application was visited during the testing period. The upper-left and upper-right bars represent the additional amount of code compiled and code executed, respectively, when Kudzu was used to explore the application. The stacked bars represent the total amount of instructions compiled and executed, where the partitions represent the absolute increase in number of instructions when Kudzu was used to explore the application.

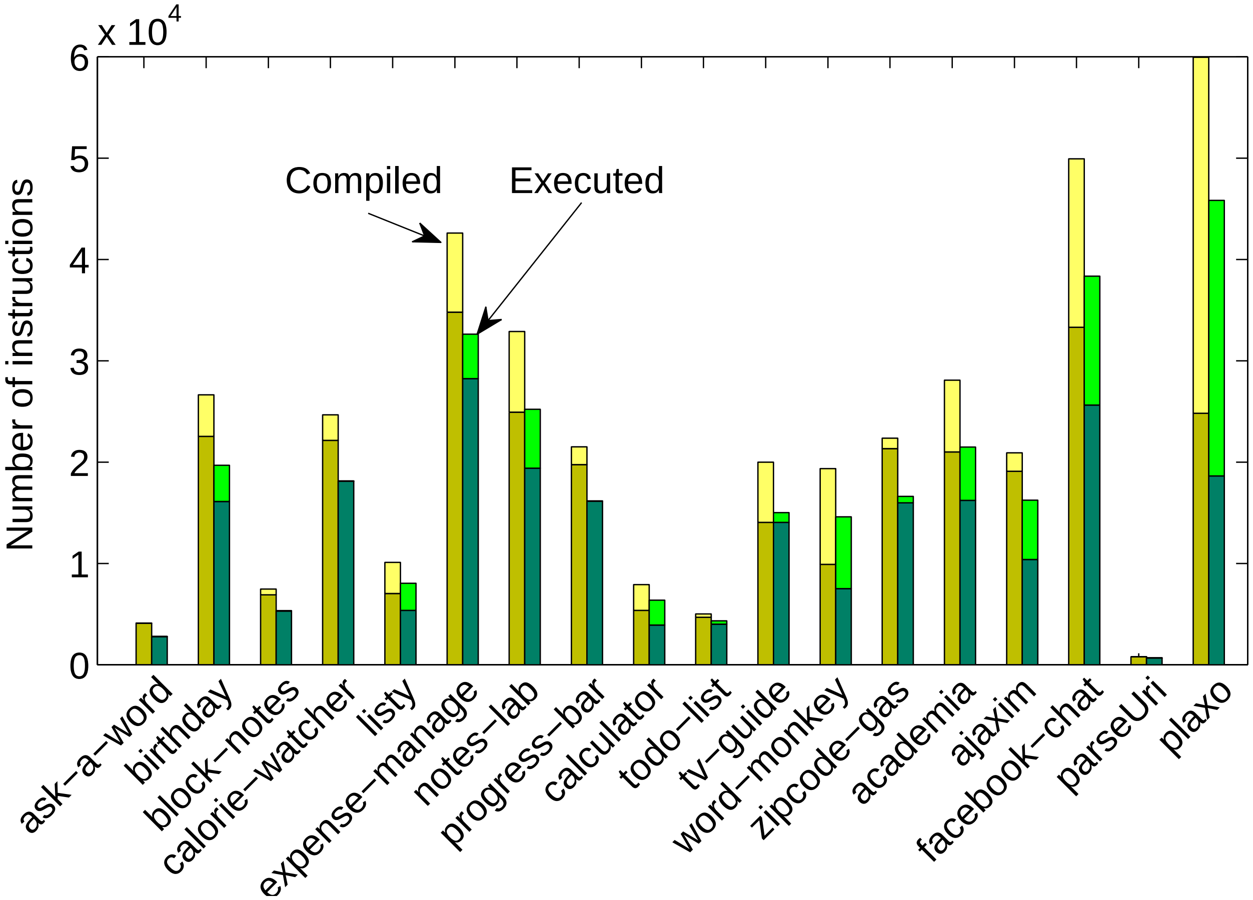


Figure 2.4: For each application, the lower-left and lower-right bars represent the amount of code compiled and executed initially when the application was executed using symbolic execution alone versus using the complete Kudzu system which includes both event and value space exploration. The upper-left and upper-right bars represent the additional amount of code compiled and code executed, respectively, when Kudzu was used to explore the application, versus only symbolically executing the application. The stacked bars represent the total amount of instructions compiled and executed, where the partitions represent the absolute increase in number of instructions.

dynamically compiled language is challenging because all the application code is not known prior to the experiments. In our experiments, we measured the total code compiled during our experiments and the total code executed [4].

Table 2.1 shows an average improvement of over 29% in code coverage. The coverage varies significantly depending on the application. Figure 2.3 provides more detail. On several large applications, such as Facebook Chat, AjaxIM, and Plaxo, Kudzu discovers a lot of new code during testing. Kudzu is able to concretely execute several code paths, as shown by the increase in the right-side bars in Figure 2.3. On the other less complex gadget applications, most of the relevant code is observed during compilation in the initial run itself, leaving a relatively smaller amount of new code for Kudzu to discover. We also manually analyzed the source code of these applications and found that a large fraction of their code branches were not dependent on data we treat as untrusted.

To measure the benefits of symbolic execution alone, we repeated the experiments with the event-space exploration turned off during the test period and report the comparison to full-featured Kudzu in Figure 2.4. We consistently observe that symbolic execution alone discovers and executes a significant fraction of the application by itself. The event-exploration combined with symbolic execution does perform strictly better than symbolic execution in all but 3 cases. In a majority of the cases, turning on the event-space exploration significantly complements symbolic execution, especially for the AJAX applications which have a significant GUI component. In the 3 cases where improvements are not significant, we found that the event exploration generally either led to off-site navigations or the code executed could be explored by symbolic execution alone. For example, in the parseUri case, the same code is executed when a user types into a text-box input or clicking a button on its GUI

Table 2.2 shows the increase in number of events executed by Kudzu from the initial run to the total at the end of test period. These events include all keyboard and mouse events which result in execution of event handlers, navigation, form submissions and so on. We find that new events are dynamically generated during one particular execution as well as when new code is discovered. As a result, Kudzu gradually discovers new events and was able to execute approximately 50% of the events it observes during the period of testing.

Effectiveness. Kudzu automatically generates a test suite of 49 new distinct inputs on average for an application in the test period (shown in column 2 of table 2.1).

In the exploitable cases we observed, Kudzu was able to show the existence of a vulnerability with an attack string once it reached the point of vulnerability. That is, its constraint solver correctly determines the sufficiency or insufficiency of validation checks in a single query without manual intervention or undirected iteration. This eliminates most false positives in practice, except in the case where our constraint language fails to model the JavaScript regular expression language in its entirety, which we discuss in Chapter 2.4.2.1.

[4] One unit of code in our experiments is a JavaScript bytecode compiled by the interpreter. To avoid counting the same bytecode across several runs, we adopted a conservative counting scheme. We assigned a unique identifier to each bytecode based on the source file name, source line number, line offset and a hash of the code block (typically one function body) compiled.

Application	# of initial events fired	# of total events fired	Total events discovered
Academia	20	78	310
AJAXIm	72	481	988
FaceBook Chat	15	989	1354
ParseUri	5	16	17
Plaxo	88	381	688
AskAWord	2	8	11
Birthday Reminder	12	20	20
Block Notes	7	85	319
Calorie Watcher	14	18	22
Expenses Manager	10	107	1473
Listy	15	470	638
Notes LP	10	592	1034
Progress Bar	8	24	36
Simple Calculator	17	34	67
Todo List	8	26	61
TVGuide	17	946	1517
Word Monkey	3	10	22
Zip Code Gas	12	12	12
Average	**18.61**	**238.72**	**477.17**

Table 2.2: Event space coverage: Column 2 and 3 show the number of events fired in the first run and in total. The last column shows the total number events discovered during the testing.

```
function validate (input) {
 //input = ’{"action":"","val":""}’;
 mustMatch = ’{]:],]:]}’;
 re1 =/\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g;
 re2 =/"[^"\\\n\r]*"|true|false|null|
      -?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g;
 re3 = /(?:^|:|,)(?:\s*\[)+/g;
 rep1 = str.replace(re1, "@");
 rep2 = rep1.replace(re2, "]");
 rep3 = rep2.replace(re3, "");
 if(rep3==mustMatch) { eval(input); return true; }
 return false;            }
```

Figure 2.5: Example of a regular-expression-based validation check, adapted from a real-world JavaScript application. This illustrates the complexity of real regular expression syntax.

An instance of false positive elimination occurred when Kudzu found that the Facebook web application has several uses of `postMessage` data in `eval` constructs, but all uses were correctly preceded by checks that assert that the origin of the message is a domain ending in `.facebook.com`. In contrast, the vulnerability in Plaxo fails to check this and Kudzu identifies the vulnerability the first time it reaches that point. Some of the validation checks Kudzu deals with are quite complex — Figure 2.5 shows an example which is simplified from a real application. These examples are illustrative of the need for automated reasoning tools, because checking the sufficiency of such validation checks would be onerous by hand and impractical by random fuzzing. Lastly, we point out that like most other vulnerability discovery tools, Kudzu can have false negatives because it may fail to cover code, or because of overly strict attack grammars.

2.4.2.1 String and Regular Expression Limitations and False Positives

Kudzu can cover the majority of regular expressions and string operations, however, our tool has a few limitations which we describe below. These limitations can lead to incorrect modeling of the underlying JavaScript string and regular expression representations, which can lead to false positives.

For instance, string replace is often used in sanitization to transform unsafe characters into safe ones. Our translation uses a concrete value for the number of occurrences of the searched-for pattern: if a pattern was replaced six times in the original run, the tool will search for other inputs in which the pattern occurs six times. This sacrifices some generality (for instance, if a certain attack is only possible when the string appears seven times). However, we believe this is a beneficial trade-off since it allows our tool to analyze and find bugs in many uses of replace. For comparison, most previous string constraint solvers do not support replace at all, and adding a replace that applied to any number of occurrences of a string (even limited to single-character strings) would make our core constraint language undecidable in the unbounded case[29].

Furthermore, the regular expressions supported by languages like JavaScript have many more features than the typical definition given in a computability textbook. Kudzu handles a majority of the syntax for regular expressions in JavaScript, which includes support for (possibly negated) character classes, escaped sequences, repetition operators ({n}/?/*/+/) and submatch extraction using capturing parentheses. Kudzu keeps track of the nesting of capturing parentheses, so that it can express the relation between the input string and the parts of it that match the captured groups. Kudzu does not currently support back-references, which are strictly more expressive than true regular expressions.

Besides these limitations, we find that Kudzu works well and gets rid of most, if not all false positives in the cases we examined.

2.5 Related Work

Kudzu is the first application of dynamic symbolic execution to client-side JavaScript. Here, we discuss some related projects that have applied similar techniques to server-side web applications, or have used different techniques to search for JavaScript bugs. Finally, we summarize why we needed to build a new string constraint solver.

Server-side analysis. JavaScript application code is similar in some ways to server-side code (especially PHP); for instance, both tend to make heavy use of string operations. Several previous tools have demonstrated the use of symbolic execution for finding SQL injection and reflected or stored cross-site scripting attacks to code written in PHP and Java [84, 53, 23]. However, JavaScript code usually parses its own input, so JavaScript symbolic execution requires more expressive constraints, specifically to relate different strings that were previously part of a single string. Additional challenges unique to JavaScript arise because JavaScript programs take many different kinds of input, some of which come via user interface events.

Like Kudzu, the Saner [25] tool for PHP aims to check whether sanitization routines are sufficient, not just that they are present. However their techniques are quite different: they select paths and model transformations statically, then perform testing to verify some vulnerabilities. Their definition of sanitization covers only string transformations, not validation checks involving branches, which occur frequently in our applications.

Analysis frameworks for JavaScript. Several works have recently applied static analysis to detect bugs in JavaScript applications (e.g., [43, 33]). Static analysis is complementary to symbolic execution: if a static analysis is sound, an absence of bug reports implies the absence of bugs, but static analysis warnings may not be enough to let a developer reproduce a failure, and in fact may be false positives.

FLAX uses taint-enhanced blackbox fuzzing to detect if the JavaScript application employs sufficient validation or not [68]; like Kudzu, it searches for inputs to trigger a failure. However, FLAX requires an external test suite to be able to reach the vulnerable code, whereas Kudzu generates a high-coverage test suite automatically. Also, FLAX performs only black-box fuzz testing to find vulnerabilities, while Kudzu's use of a constraint solver allows it to reason about possible vulnerabilities based on the analyzed code.

Crawljax is a recently developed tool for event-space exploration of AJAX applications [61]. Specifically, Crawljax builds a static representation of a Web 2.0 application by clicking elements on the page and building a state graph from the resulting transitions. Kudzu's value space exploration complements such GUI exploration techniques and enables a more complete analysis of the application using combined symbolic execution and GUI exploration.

Artemis is a tool developed by IBM for feedback driven, random testing of JavaScript applications[22]. Artemis explores a prioritized space of events and inputs. New inputs are generated by modifying previous test inputs, and new events are discovered by executing the same application again but modifying inputs and event sequences in order to gain code coverage. Primarily, they avoid reasoning about JavaScript string constraints in favor of a simpler, yet effective, method of randomly exploring JavaScript applications. However, Artemis will miss code paths in applications which have complex checks based on a string's format.

2.6 Conclusion

With the rapid growth of AJAX applications, JavaScript code is becoming increasingly complex. In this regard, security vulnerabilities and analysis of JavaScript code is an important area of research. In this chapter, we presented the design of the first complete symbolic-execution based system for exploring the execution space of JavaScript programs. In making the system practical we addressed challenges ranging from designing a more expressive language for string constraints to implementing exploration and replay of GUI events. We have implemented our ideas in a tool called Kudzu. Given a URL for a web application, Kudzu automatically generates a high-coverage test suite. We have applied Kudzu to find client-side code injection vulnerabilities and Kudzu finds 11 vulnerabilities (2 previously unknown) in live applications without producing false positives.

2.7 Acknowledgments

This chapter, *A Symbolic Execution Framework for JavaScript* originally appeared at **Oakland 2010**. This is an original work by Prateek Saxena, Devdatta Akhawe, Steve Hanna, Stephen McCamant, Feng Mao and Dawn Song. We thank David Wagner, Adam Barth, Vijay Ganesh, Adam Kieżun, Domagoj Babic, Adrian Mettler, Juan Caballero and Pongsin Poosankam for helpful feedback on this chapter.

Chapter 3

The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives

3.1 Introduction

Several new browser primitives have been proposed to meet the demands of application interactivity while enabling security. To investigate whether applications consistently use these primitives safely in practice, we study the real-world usage of two client-side primitives, namely `postMessage` and HTML5's client-side database storage. We examine new purely client-side communication protocols layered on `postMessage` (Facebook Connect and Google Friend Connect) and several real-world web applications (including Gmail, Buzz, Maps and others) which use client-side storage abstractions. We find that, in practice, these abstractions are sometimes used insecurely, which leads to severe vulnerabilities and can increase the attack surface for web applications in unexpected ways. We conclude the paper by offering insights into why these abstractions can potentially be hard to use safely, and propose the *economy of liabilities* principle for designing future abstractions. The principle recommends that a good design for a primitive should minimize the liability that the user undertakes to ensure application security.

Summary of Contributions[1].

- We systematically examine two representatives of new client-side primitives which are in popular use by real-world applications: (a) `postMessage`, a cross-domain

[1] **In this chapter, I was jointly responsible for the idea to study HTML5 primitives, as a natural extension of our previous web work, I conducted the clientStorage experiments and worked on suggestions for improving these primitives.**

message passing API, and (b) persistent client-side database storage (HTML5 localStorage, webDatabase APIs and database storage in Google Gears).

- We present the first step towards understanding purely client-side protocols, by reverse engineering them directly from their implementation in JavaScript and formalizing them. We systematically extract the sanity checks that applications implement on the security-relevant data and use these to find new vulnerabilities in our target applications.

- We provide practical evidence of the pervasiveness of these new attacks on several important web application protocols (Facebook Connect and Google Friend Connect) and web applications (Gmail, Google Buzz, Google Docs and others).

- To eliminate the variation we observe in safe usage of these client-side primitives, we propose the guiding principle of *economy of liabilities* and suggest remedies based on this principle to make the primitives more practical for safe use with the aim of garnering discussion and obtaining community feedback. Our experiments indicate that developers sometimes know how to use the primitives, and even make deliberate decisions to ignore the security properties offered and implement their own protocol on top of these new primitives.

3.2 Attacks on Client-side Messaging

The postMessage API is a client-side primitive to enable cross-origin communication at the browser side. Originally introduced in HTML5, postMessage aims to provide a simple, purely client-side cross-origin channel for exchanging primitive strings[79]. Web browsers typically prevent documents with different origins from affecting each other [75]. A mashup specifically aims to overcome this restriction and communicate with another web site in order to provide a richer experience to the user. Barth et al.[26] study various client-side cross-origin communication channels and recommend the postMessage mechanism, due to the security guarantees (detailed below) it is able to provide.

The postMessage primitive aims to provide the dual guarantees of authenticity and confidentiality. Messages can be sent to another window by invoking the window's postMessage method. Note that this message exchange happens completely over the client side and no data is sent over the network. The security guarantees are achieved as follows:

- *Confidentiality*: The sender can specify the intended recipient's origin in the postMessage method call. The browser guarantees that the actual recipient's origin matches the origin given in the postMessage call, and code executing in any other origin's context is unable to see the message. The intended recipient's origin, specified in the method call, is called the targetOrigin parameter. For use cases in which confidentiality is not essential, a sender can specify the all-permissive '*' literal as the targetOrigin.

- *Authenticity*: The browser attributes each received message with the origin of the sender, as the `origin` property of the message event. The recipient is expected to validate the sender's origin as coming from a trusted source, thus achieving sender authenticity.

Note that if these checks are missed by the application, the browser does not guarantee anything about the security of the `postMessage` channel. For instance, a malicious website could send arbitrary messages to a benign website, and it is the latter's responsibility to ensure that it only processes messages from trusted senders. To avoid the aforementioned problems, the HTML5 proposal recommends websites to set the `targetOrigin` parameter for any confidential message and to always check the `origin` parameter on receiving a message.

Attacking postMessage Applications. We investigate two prominent users of the `postMessage` primitive, the Facebook Connect protocol and the Google Friend Connect protocol. We conjecture that for complex cross-domain interactions involving fine-grained origins, developers may fail to follow the recommended practice. In such a case, the channel would not provide a security property that the developer might have come to expect. Due to the complexity of the JavaScript code used by these protocols, we use the Kudzu [67] system to check for the absence of such validation in the code. We find that large parts of the protocols are undocumented, and we reverse engineer these protocols based on the interactions we observe.

Scope of Attack. The threat model for our attacks on `postMessage` usage is the web attacker threat model [27]. In particular, we constrain the attacker to only controlling content on his own site. A user can visit the attacker's site, but may not necessarily trust content from it. Phishing attacks are outside the scope of this work. Bugs in browser implementations are also beyond the scope of this attack. An attacker can assume the user to have already logged onto Facebook and authorized Facebook Connect applications not controlled by the attacker.

Summary of Findings. We find various inconsistencies in the use of `postMessage`. Developers use these primitives correctly in some cases, while making mistaken assumptions in others. We demonstrate vulnerabilities in both Facebook Connect and Google Friend Connect protocols. We explain these two protocols in detail, point out vulnerabilities and demonstrate concrete attacks. We end our analysis of the `postMessage` primitive with a discussion of the observed real world usage of the `postMessage` primitive.

3.2.1 The Facebook Connect protocol

Facebook Connect is a system that enables a Facebook user to share his identity with third-party sites. Some notable users include TechCrunch, Huffington Post, ABC and Netflix. After being authorized by a user, a third party web site can query Facebook for the user's information and use it to provide a richer experience that leverages the user's social connections. For example, a logged-in user can view his Facebook friends who also use the third-party web site, and interact with them directly there. Note that the site now contains content from multiple principals—the site itself and `facebook.com`.

Mechanism. The same-origin policy does not allow a third-party site (e.g TechCrunch), called `implementor` in the paper, to communicate directly with `facebook.com`. To support this interaction, Facebook provides a JavaScript library for sites implementing Facebook Connect. This library creates two hidden iframes with an origin of `facebook.com` which in turn communicate with Facebook. The cross-origin communication between hidden iframes and the implementor's window are layered over `postMessage`[2].

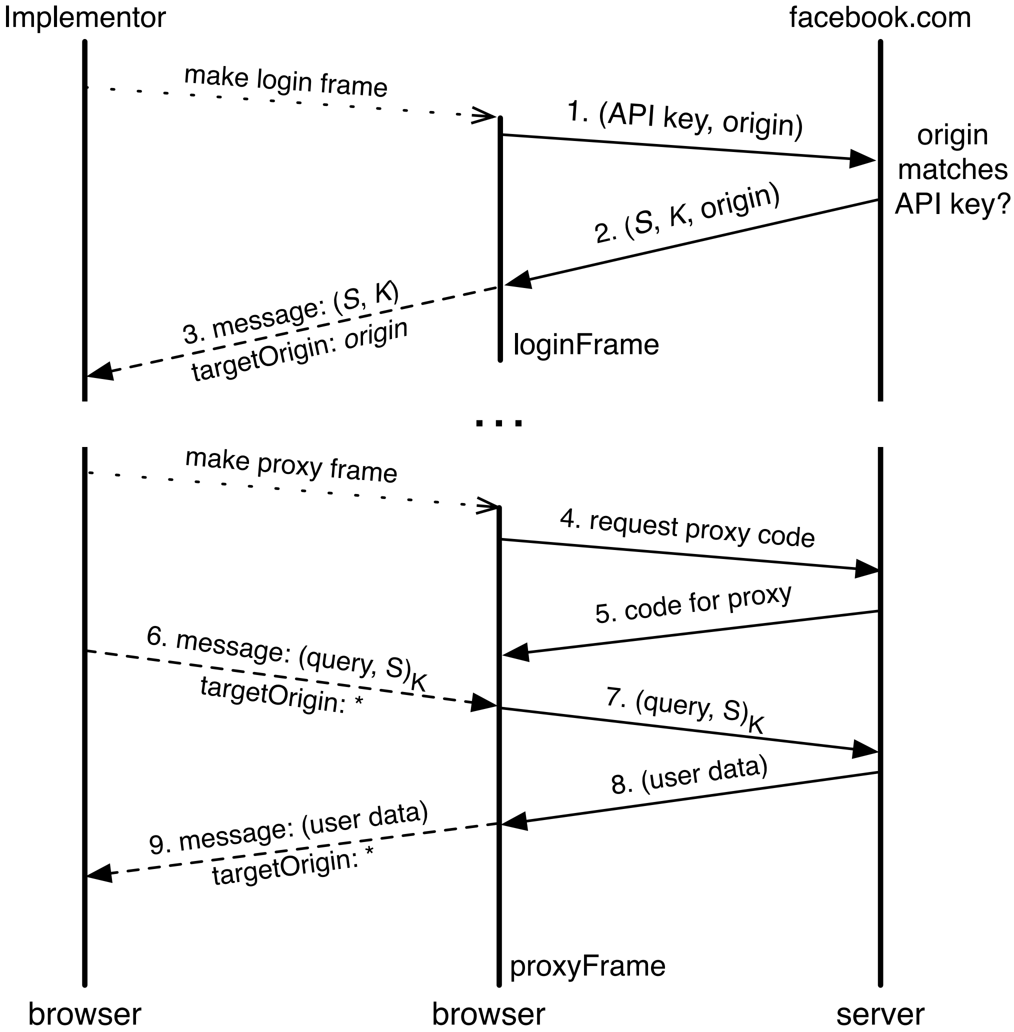


Figure 3.1: The Facebook Connect protocol. Ladder diagram of messages exchanged in the protocol. The dashed arrows represent client-side communication via `postMessage` and the solid arrows represent communication over HTTP. `(query,S)`$_K$ represents a HMAC using the secret K. Frame hierarchy for the Facebook Connect protocol. In this example, the `proxyFrame` is inside the main `implementor` window.

[2] In older browsers, other techniques are used which we do not discuss.

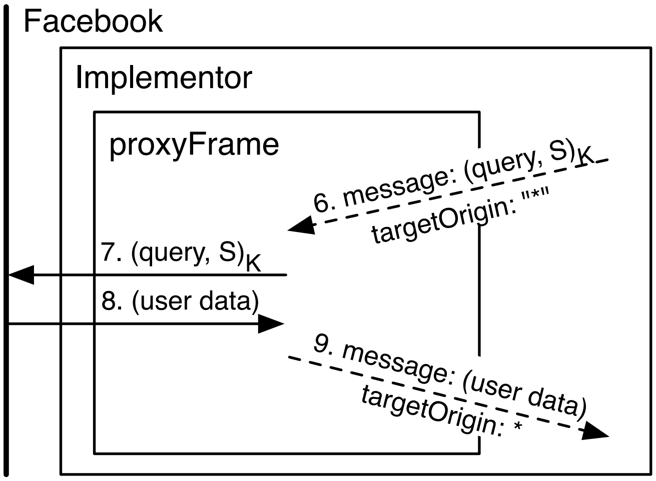


Figure 3.2: The Facebook Connect protocol. Frame hierarchy for the Facebook Connect protocol. In this example, the `proxyFrame` is inside the main `implementor` window.

Figures 3.2 and 3.1 detail the protocol. The first iframe created by the library is used for the initial session negotiation with Facebook and the other is used for all subsequent data exchanges between the Facebook server and its client-side counterpart. More specifically, the first iframe (`loginFrame`, in Fig 3.2) receives a secret key (K) and a session ID (S) from `facebook.com` and sends it to `implementor` (message 3). The second iframe (`proxyFrame`, in Fig 3.2) also running in `facebook.com`'s origin, acts as a proxy for requests. Any query for data that `implementor` wants to make to `facebook.com` is first sent to `proxyFrame` (message 6), which then makes the request to `facebook.com` using `XMLHttpRequest` (message 7) and then sends the response (message 8) back to `implementor` (message 9). At the end of this transaction, the user has essentially logged in to `implementor` using his Facebook credentials.

3.2.2 Vulnerabilities in Facebook Connect

Observation 1: During our testing, we noticed that the `origin` of received messages was sporadically verified. In particular, out of all of the messages exchanged, only about half were accompanied with an origin check in the receiver's code. Further investigation revealed that neither participant, namely the `proxyFrame` and the implementor (message 6 and 9), checked the origin of received messages.

Additionally, we also noticed that the message 6 and 9 had the `targetOrigin` parameter set to the '*' literal, while in message 3, the `targetOrigin` parameter was correctly set. We also observe that a query for data is authenticated by an HMAC with the shared secret K. This serves as a signature for every query (message 6) that the `proxyFrame` receives.

Attack on message integrity. As discussed before, validating the origin of received messages is necessary for ensuring sender authenticity. Based on *Observation 1*, a malicious attacker can inject arbitrary data in the communication between `proxyFrame` and `implementor`. In this particular case, we find that the data received over the channel is used in a code evaluation construct and thus allows an attacker to inject arbitrary code into `implementor`'s security context.

The attack is illustrated in Figures 3.3 and **??**. In particular, an attacker replaces `proxyFrame` with a malicious iframe that he controls. By sending a malicious message in place of message 9, an attacker can inject a script into the implementor's security context. In the actual attack, the attacker has to include the `implementor` page in a iframe on a page controlled by him (see bottom of Figure 3.3). This gives the attacker the power to replace the benign Facebook `proxyFrame` with his own malicious `proxyFrame`. This attack is possible because on receiving message 9, the implementor does not validate the origin of the message sender, and thus processes a message from the attacker. The shared secret only provides authenticity of the query (message 6) and not for the response (message 9).

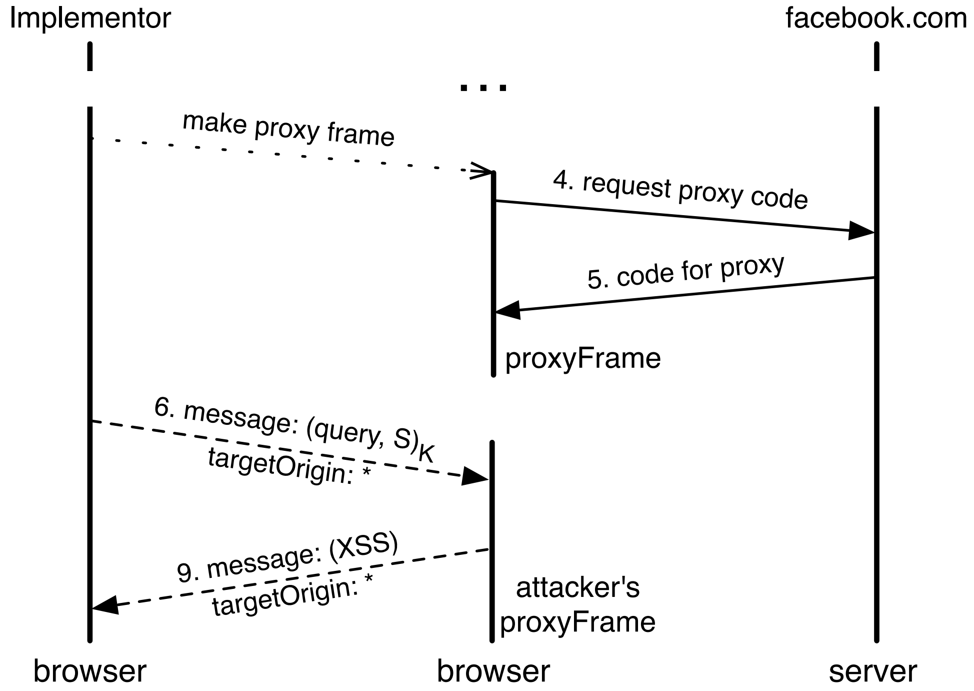


Figure 3.3: Integrity attack on Facebook Connect. Messages exchanged in the protocol. Note that midway through the protocol (after message 5), the request proxy is replaced by an attacker-controlled proxy.

On our test site, we were able to inject a script payload into the benign implementor's security context. We had previously discovered a similar flow of data to a critical code evaluation construct, which was fixed by Facebook by adding data sanitization routines [67]. This is not a scalable fix. We have also confirmed this attack on Facebook's reference implementation of a Facebook Connect site. As the Facebook Connect functionality is provided as a drop-in JavaScript library, we believe most real-world websites directly using this library are also vulnerable.

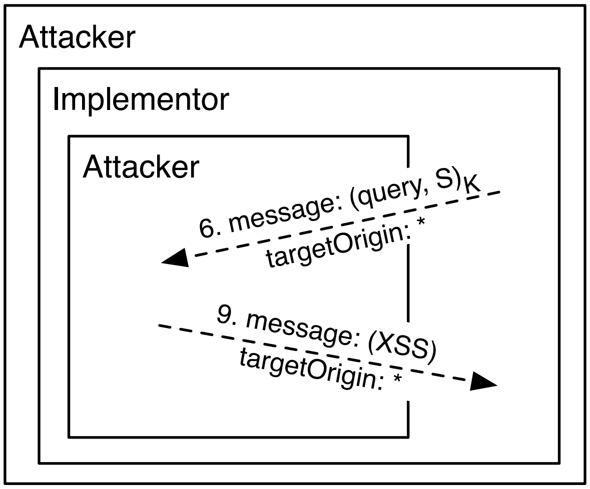


Figure 3.4: Integrity attack on Facebook Connect. Frame hierarchy for the integrity attack. The topmost frame is owned by the attacker.

Attack on confidentiality. *Observation 2*: Setting the `targetOrigin` parameter to the '*' literal leaks sensitive user data like profile information and friend lists to the attacker. This data can then be used by the attacker to gain the real-world identity of a visitor to his website.

The attack is illustrated in Figure 3.5 and 3.6. Message 9 and Message 6 have the `targetOrigin` set to '*'. Based on *Observation 2*, this allows a malicious attacker to easily launch a man-in-the-middle attack against the communication between the `implementor` and the `proxyFrame` (message 6a in Fig 3.5). The fact that `implementor` does not validate the sender of messages (of message 9a in Fig 3.5, in particular) enables a complete man-in-the-middle attack, while the signature on the query provides no protection. The main attack occurs at message 9 (Fig 3.5), which consists of sensitive user data and is read by the attacker. In the actual attack, the attacker again includes the benign `implementor` page in an iframe and then replaces the `proxyFrame` with his man-in-the-middle frame, which in turn includes the real `proxyFrame` (bottom of Fig 3.5).

3.2.3 The Google Friend Connect protocol

Google Friend Connect is a system that provides similar functionality to Facebook Connect. An important difference is that Google Friend Connect allows a user to use multiple identity providers (like Yahoo!, Twitter, or Google) while signing onto various third-party sites. The aim, again, is to enable a richer social experience for users.

Mechanism. Typically, Google Friend Connect applications embed 'gadgets' inside iframes, which directly communicate with the relevant server. These gadgets communicate with the integrating page, referred to as `implementor` in the paper, via `postMessage` for parameters like colors, fonts and layouts. Like Facebook Connect, third-party websites

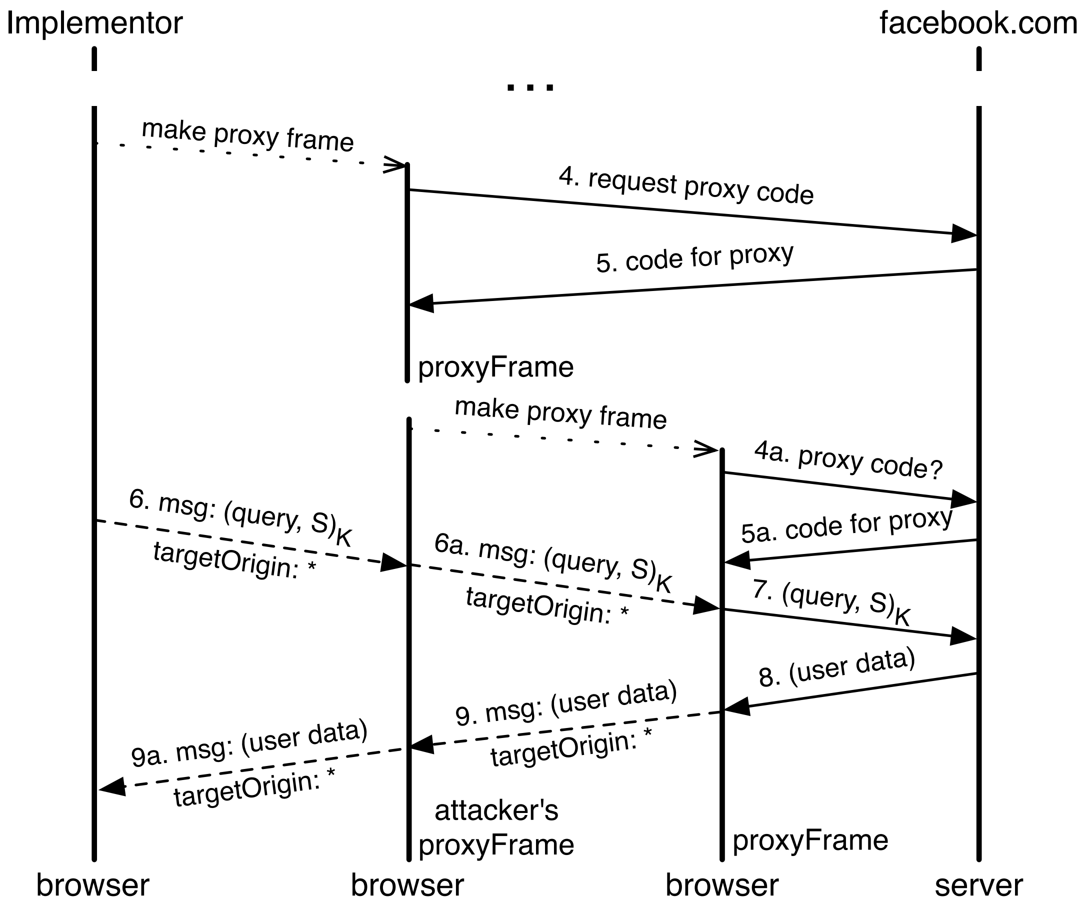


Figure 3.5: Confidentiality attack on Facebook Connect. Message Exchange—note the replayed messages 6a and 9a.

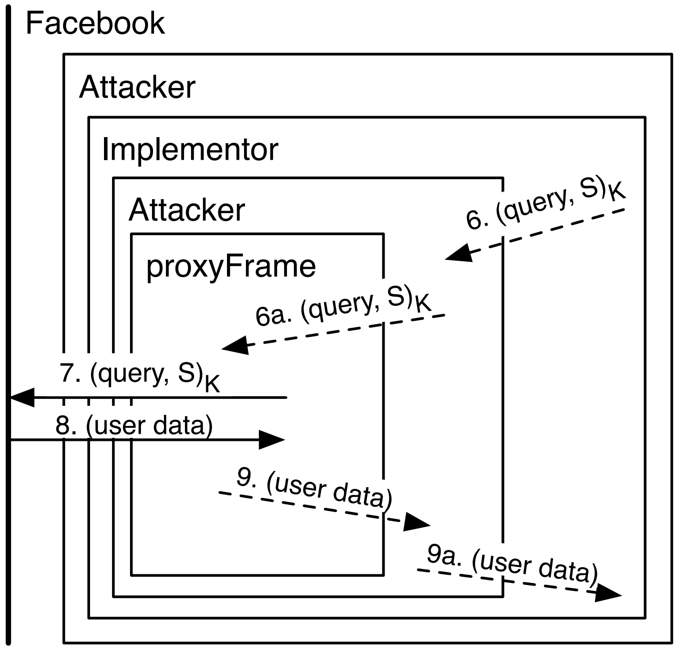


Figure 3.6: Confidentiality attack on Facebook Connect. Frame hierarchy for the confidentiality attack. Note the presence of two attacker frames—the main window frame and the man-in-the-middle frame.

interested in integrating Google Friend Connect in their sites need to include a Google JavaScript library in their pages.

Figure 3.7 details the protocol. The code running in the `implementor`'s context generates a random nonce (N), and creates an iframe that requests a gadget (message 1 in Fig. 3.7). The nonce is included in the request as a GET parameter. Subsequent communication (messages 4 and 5) between the gadget and the `implementor` includes this nonce. Notice that the private user data `(user info)` is never sent over a `postMessage` channel.

3.2.4 Vulnerabilities in Google Friend Connect

Observation 3: During our testing, we noticed that all message exchanges in the Google Friend Connect protocol had the correct `targetOrigin` set. Analysis of the JavaScript code revealed the absence of any sender authenticity checks. In particular, for all the 12 messages that were exchanged, no participant checked the message sender's origin. Instead, we noticed checks for the nonce (N in Fig. 3.7). The protocol uses the nonce to authenticate all message exchanges. As the `targetOrigin` is correctly set for all messages, the nonce can never leak to an attacker.

Observation 4: The random number generator provided by the browser (via `Math.random`) is not cryptographically secure (as shown in [57]). With just one call to `Math.random()`, an attacker can guess all future values of `Math.random()`. This

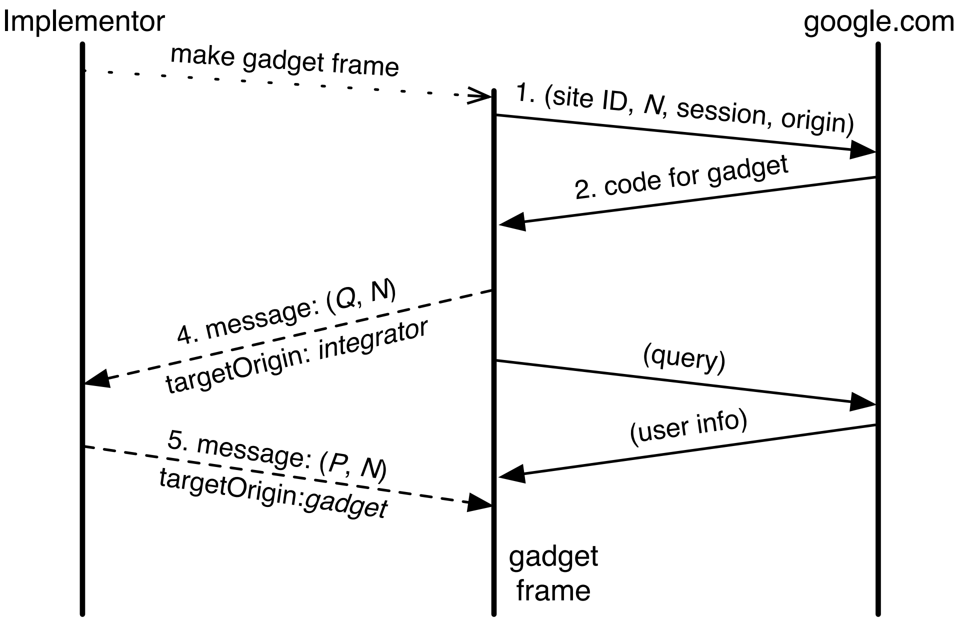


Figure 3.7: Google Friend Connect's gadget protocol: the nonce N is generated by the `implementor`. Message 4 is a query Q for parameters. The `implementor` responds with the parameters P in message 5.

breaks the authentication used by Google Friend Connect. For example, on Firefox 3.6, we were able to exactly predict the nonce that would be used by the Google Friend Connect protocol.

Similar to the Facebook Connect attack, the attacker can embed the benign `implementor` in an iframe within his own malicious page. The attacker's page can then sample `Math.random()` to predict the value of the nonce, and then spoof any message exchanged by `implementor` and the gadget over `postMessage`, compromising the Google Friend Connect session (see figure 3.7). Based on *Observation 3* and *Observation 4* we observe that this attack would have failed if the Google Friend Connect protocol validated the message sender by checking the `origin`, rather than relying on predictable nonces. Correctly setting the `targetOrigin` on all messages makes the protocol secure against confidentiality attacks.

3.2.5 Discussion

Authenticity and confidentiality are strong properties that the `postMessage` API can provide, in principle. Our study of real world usage of the `postMessage` API reveals that developers sometimes do not use the abstractions provided by the `postMessage` primitive correctly. Designing in-house secure protocols is challenging—as we've seen. Both Facebook Connect and Google Friend Connect tried to achieve sender authenticity by using their own system (secret nonce or HMAC), instead of the recommended practice (checking the `origin` parameter). We were able to circumvent the authentication methods used by these protocols and insert malicious messages in the communication. In the case of Facebook Connect, we were also able to achieve arbitrary code execution.

3.2.5.1 Potential Difficulties with postMessage

We first conjecture one possible cause of the errors we discovered. Then, we include a discussion with Facebook regarding their use of `postMessage` which indicates that the vulnerabilities were caused by an unrelated problem. We discuss both below.

Consider this example:

```
if(event.origin == ‘http://example.com’) {
    // execute code
}
```

Such examples give a false sense of simplicity. In the real world, the source of messages could be one of many possible fine grained origins. So, writing checks can involve the creation of a long and potentially complex regular expression and string operations, especially when multitudes of origins are permissible. Additionally, for complex protocols, these checks must be repeated for every message— which can lead to code bloat, repeated checks, and can be easily forgotten as permissible origins are added or deleted from the application. Furthermore, the complexity of the code increases if different code is executed depending on the chosen origin.

```
eo = event.origin
var re1 = /https?:\/\/[abc].example.com/;
var re2 = /http:\/\/examplepartner.com|org/");
var re3 = /http:\/\/[xyz].otherexamplepartner.com|org/");
var re4 = /https?:\/\/example.com/");

if (re1.test(eo) || re2.test(eo) || re3.test(eo) ||
re4.test(eo) || ... || reN.test(eo)
/*more checks for each valid origin*/ ) {
//execute code
}
```

Implementing this functionality would require a series of string-based verification comparisons, namely complex regular expressions—which is precisely the problem we have exemplified above. Furthermore, if a mashup includes content from more than a couple of origins, these checks become even more taxing. Fundamentally, this is a usability issue of the API. In Chapter 3.4, we suggest potential enhancements to the specifications to mitigate these issues, in keeping with the *economy of liabilities* principle.

The use of the all-permissive ‘*’ as the `targetOrigin` allows leakage of confidential data. The HTML5 specification [79] warns against the use of the ‘*’ literal for confidential data. We believe giving developers the choice of insecure usage is not a good practice. Additionally, it is notoriously hard to figure out what data is privacy sensitive and what isn't[63]—and we believe this will only get more difficult. Based on these facts, we suggest a possible modification in Chapter 3.4.

Vendor Discussion. We conjectured the developers created their own network style protocol due to the fact that 'simple' sender origin checks may not be easy to get correct[3]. However, our discussion with Facebook regarding Facebook Friend Connect yielded edifying insights into the compatibility issues that vendors have to deal with —especially, those arising from the same client-side code having to behave the same in a variety of browser execution environments. Specifically, we asked Facebook why they chose to ignore the origin field when receiving messages over `postMessage`. Facebook responded that only about half of their users' browsers support `postMessage` and neither Flash nor fragments provide the origin, as required for authenticity checks. In response to this, they developed the protocol we previously discussed that relies on a nonce for authenticating messages. In fact, in our discussion with Facebook, we were informed that they used the all-permissive '*' directive because `postMessage` does not support multicast. Multicast, in this sense, would implement a method in which a message could be sent or received to or from a list of list of permissible origins. Furthermore, we discussed the idea of *multicast* enabled messaging, which we discuss in Chapter 3.4, with Facebook and they responded that they would find that modification to the specification to be a useful addition.

Additionally, we contacted the security team at Google to notify them of the vulnerabilities we found in Google Friend Connect. Google responded by stating that the Friend Connect team was investigating the issue and it is currently fixed.

3.3 Persistent, Server-Oblivious Client-side Database Attacks

In this chapter, we study the usage of new client side persistent storage mechanisms supported by HTML5 and Google Gears. We find that data stored in client-side databases is often used in code evaluation constructs without sanitization. Client-side databases, thus, provide additional avenues for attackers to persist their payloads across sessions. For instance, attackers only need to inject XSS attack payloads once into the client-side storage to have them repeatedly compromise the client-side code integrity for sustained periods of time (unlike a common reflected XSS issue which is fixed once patched). Additionally, because the attack payload is stored on the client-side, the server is oblivious to the nefarious activity. We show that the 7 out of 11 major applications we studied trust the client-side storage and are vulnerable to such *persistent* attacks, including: Gmail, Google Buzz, Google Documents and others.

[3] Perhaps due to the complicated regular expressions required to match all possible origins able to send messages to the implementor.

3.3.1 Client-side Storage: Background

HTML5 proposes two persistent storage abstractions: `localStorage` and `webDatabase`[81, 80]. A limited number of browsers currently support these features. The client-side storage mechanisms work as follows:

- `localStorage` is a key/value store tied to an application's origin. Script executing within an origin can get or set values of the data store using the `localStorage` object.
- `webDatabase` is a client-side database that supports execution of SQL statements. The database is bound to the origin in which the code executes and web applications are restricted to only modifying the structure and contents of their associated origin's database. To execute SQL against the database one can use: `executeSql(sqlStatement, arguments, callback, errorCallback).`
- Gears is a Google product designed to enable applications to work offline. Recently, Google has decided to deprecate Gears in favor of HTML5[38]. Despite syntactic differences, Gears and HTML5 `webDatabase` data storage work in very similar ways.

In each of these cases, database modifications persist until the creating application destroys the data.

3.3.2 Persisting Server-Oblivious Attack Payloads

We consider two possible attack vectors in our threat model, a network attacker and a transient XSS vulnerability.

The goal of either attacker is to inject code into the local storage in order to gain a persistent foothold in the application—one that remains even when the transient attack vector is fixed. Once an application has been compromised, the attacker has control of the application until the client side database is cleared. In current implementations, this only occurs when the database is explicitly cleared by an application, making the attack have a long lifetime.

Network Attacker. Consider the case when a network attacker is able to modify packets destined to the victim. When the user visits a site using client-side storage the attacker modifies the victims network packets to allow the network attacker to inject arbitrary JavaScript. This allows the attacker to compromise the database with no trace server-side that a client-side exploit has occurred until the client-side database is cleared. However, in the case of a network attacker, it is worth pointing out that other attacks may be more fruitful to the attacker such as: a full man-in-the-middle attack or Cross-Channel Scripting[28]. Both of these attacks can persistently inject content at the network and application level respectively, which pose greater persistent threats.

As an example of a realistic scenario, consider when a user visits a coffee shop with open wireless. Unbeknownst to him, the network attacker intercepts his network connections so that they are forwarded through the attacker's computer. When the user visits Google Buzz, the network attacker modifies the page returned to supply a script which modifies the client-side database. Now, whenever the data from the database is used in a code evaluation construct, the attack payload is executed instead. The user now leaves the cafe with a compromised machine and due to the stealthy injection (with no server side XSS required), little evidence remains that an attack occurred.

Transient XSS. As a second attack vector, suppose that an attacker has exploited a transient XSS vulnerability as a primary attack vector and has been able to execute arbitrary code within the context of the target site. The attacker is able to modify the database arbitrarily because the attacker has used the XSS to execute JavaScript with the same privilege as the code running within that origin. Not only is this attack persistent, it is also *stealthy*. Besides the initial XSS injection vector, all of the code execution and state modification happens on the client-side rendering the server oblivious to the attack. It is worth noting that if the attacker has a primary injection mechanism, one that is not transient, then the primary XSS is a strictly more powerful due. This is due to the ability to perform tasks on the behalf of the user from within the context of the web application.

For a concrete example, suppose an attacker finds an XSS attack on a web email application that uses `webDatabase` to save emails. In such a case, the attacker writes an exploit such that its payload is stored inside an email in the database. When the user views the email, the injected code is executed. Now, even if the XSS vulnerability is fixed, the payload persists as long as the database.

In either case, it's important to note that if the injected database data is used in code evaluation constructs, such as `eval` or `document.write` without proper sanitization (as we observed), the attack can persist its attack payload. This payload can be used for a variety of attacks such as stealing passwords, cookies and email. The execution of the code on the client-side and resulting payload is *stealthy* because the server is oblivious to the compromise.

3.3.3 Approach

We evaluated **11** applications that use client-side storage using Kudzu. Kudzu, a systematic vulnerability finding tool built on the WebKit framework, is a dynamic symbolic execution engine framework which is designed to analyze JavaScript applications running in browsers[67]. We modified Kudzu to mark database outputs as symbolic and we note a possible vulnerability when a database output flows to a critical sink (like `innerHTML` or `eval`). All vulnerabilities were verified in Safari 4.0.4 by modifying the content of the database being targeted to contain executable code. Experiments using Google Gears were verified in Firefox 3.5.8. We verify that the code is executed by viewing the target application. In order to ensure that HTML5 features were used when applicable, we modified our `User-Agent` string to match the latest reported by an Apple iPhone.

Experimental Results. Figure 3.8 shows that we find vulnerabilities in **7** applications. In addition, it presents the type of persistent storage being used, and whether or not the database modification remains persistent.

Application	**Storage Type**	**Vulns.**	**Persistent?**
Gmail	Database	Yes	Yes
Google Buzz	Database	Yes	Yes
Google Calendar	Database	No	N/A
Google Documents	Gears	Yes	Yes
Google Maps	Database	Yes	Yes
Google Reader	Gears	Yes	Yes*
Google Translate	Database	No	N/A
Snapbird	localStorage	Yes	Yes
Remember The Milk	Gears	No	N/A
Yahoo Apps Mobile	Database	No	N/A
Zoho Writer	Gears	Yes	Yes*
Total	—	—	**7**

Figure 3.8: A security evaluation of applications using client-side database storage. The modified database persisted through reloading of the application, closing the browser, and logging in and logging back out. Note: (*) indicates that the attack only persisted while the application was in offline mode.

Gmail. We walk through a sample attack on Gmail to give an idea how a typical persistent attack may take place. First, we launch Gmail using Kudzu to analyze the application. We login to our account and are then taken to our Inbox. After this we close the browser. Kudzu then notifies us that it found data going from the database into the inner text of a `div` tag, without proper sanitization.

We concretely verified the attack. First, we note that Safari implements an SQLite database on a per origin basis. We open the database associated with Gmail, in this case `/Library/Safari/Databases/https mail.google.com_0`, and modify the `body` field of message found in the `cached_messages` table to include the text `<img src=dne onerror=alert(1);>`. When the Gmail application uses the database, the cached message containing the attack payload is executed.

3.3.4 Discussion

Our experimentation reveals variation in the way that developers sanitize their database outputs before using them in critical constructs. We found that many prominent applications, such as Google Reader, Gmail and Google Buzz do not sanitize their database output at all.

In contrast, we found a few applications aware of the severity of the mentioned attacks and they perform some kind of sanitization on their database output.

One such application, Google Calendar, sufficiently mitigates the attack. It uses a complex combination of JSON and XML to verify the data format, and sanitizes the user input to further ensure that scripts were not injectable.

Another application that mitigates code injection is Google Translate. When using Translate, the result of a translation is placed into a text node on the user's page. Therefore, the attack is mitigated as no code can be executed in a text node.

However, all of the other applications failed to sufficiently sanitize database outputs. We speculate that some applications did not sanitize database outputs because of the complexity of the sanitization process required to eliminate the attack. Alternatively, the developers may be confident enough with their sanitization prior to inserting it into the database that they omit output sanitization. Consider Gmail and Google Buzz, two applications that have fields in their database representing the textual content of an email or buzz respectively, in both cases, containing HTML. When these fields are modified by an attacker, the original content and injected attack text are rendered to the user, without the attack text being sanitized. In Gmail and Buzz, the textual content is mixed with HTML and the task of stripping away all of the possible scripting elements which result in code execution is difficult. Thus, when an attacker views the email or buzz, the persistent code in the database executed[4].

We also found some intermediate cases, including Zoho Writer, a web browser based document editor, and Google Reader. Both applications were only susceptible to a transient client-side database attack. That is, the data only persists in the offline store for as long as the client was offline. When the user returned online, the cache was cleared and refreshed with new content.

These examples show that different applications vary in the richness of content that they store in the database. Furthermore, developers, even within the same organization, may choose different security strategies when storing content, which could lead to potential problems when applications interact.

3.3.4.1 Vendor Discussion

We contacted the security team at Google to discuss the persistent, client-side database attacks that we discovered against a variety of their applications. Google responded to our vulnerability disclosure stating that their primary concern is preventing users from primary XSS attack vectors. They added that they felt, to a large extent, that these problems were inherent to the way that the mechanisms such as `localStorage` were designed and that first-degree XSS attacks are their primary focus.

[4] As of January 12, 2010, GMail supports always on SSL[69], thus breaking this version of the attack unless the client has explicitly turned off SSL-enabled GMail.

3.4 Enhancements

Client-side browser primitives expect users to perform multiple sanitization checks at various points in the code, to prevent the attacks we outlined. Further, such validation functionality is duplicated across applications. These checks are tedious, repetitive and sometimes complex, which adds unnecessary liability to developers leading to inconsistencies in use and errors. In Chapter 3.1, we proposed the general principle of economy of liabilities in the design of abstractions which helps minimize the required liability on users to ensure security.

Retrofitting the principle in existing client-side primitive designs is challenging. Below we suggest enhancements to the primitives we study, in ways which are a compromise between the need for flexibility, compatibility and security.

3.4.1 Enhancing postMessage

In Chapter 3.2, we raised the question of whether it should be possible to make the `postMessage` design easier for safe usage. We believe this is a topic of debate for the web community, in light of the empirical fact that early adopters of `postMessage` are using the primitives unsafely. On the flip side, we point out that any changes to the web platform come with cost to compatibility and generality too. We outline our suggestions below to stimulate the discussion on the best way to use these primitives securely.

Origin Whitelist. Based on the current usage, in order to ensure authenticity of messages received, we suggest a declarative system for specifying origins allowed to send messages will function better than manual origin checks. For instance, the Content Security Policy proposal allows a website to specify a whitelist of origins trusted to execute code in the website's security context [62]. We suggest extending CSP with a directive to specify origins allowed to send messages to the website. Moreover, the CSP proposal has gone through intense community discussion and at least one implementation—making it a potential starting point to build on.

In addition, from our experiments and evaluation of applications that use the `postMessage` API, in order to protect confidentiality, we recommend that broadcast should be disabled in favor of *multicast*, based on our discussions with Facebook during responsible disclosure. Currently, `postMessage` does not permit wildcard characters in domain names. However, to support multicast the API could be changed to allow the application to declaratively specify a wildcard in a domain name (e.g. `*.facebook.com`). This would restrict the domains capable of receiving messages without the need for complex regular expressions for parsing and verification. Additionally, if required, allowing for finer-grained control for recipients is also a possibility—the `postMessage` function could take a list of origins that are allowed to receive the specified message. With this primitive in place, it would be the *browser*'s responsibility to check the sender's origin with this whitelist before delivering the message.

Origin Comparison Primitive. Instead of requiring every user of the `postMessage`

API to implement a function for comparing origins, it would be much more efficient for the browser to provide this as a primitive function. If the browser provided the primitive, such a function would support comparison based on some standard language for specifying origins (like the grammar in CSP [62]). Note that browsers already have to do such checks for enforcing the same origin policy [75]. The grammar for this list could be similar to the grammar for origins specified in Content Security Policies, omitting the all-permissive '*' [62].

3.4.2 Pre or Post Sanitization of Database Content

Sanitizing the values stored by a database before using them in critical constructs can protect against persistent XSS attacks. We found few applications which performed any type of database output sanitization. But, like `postMessage`, we noticed that the output sanitization can often be complex and occur throughout the application code. Developers have two choices, either implement functionality to sanitize their data in such a way that they are confident that when it is stored it does not contain any malicious code. Or developers must be confident in their expected format when pulling data from the database such that they can be sanitized and verified before presenting this data to the web application.

We realize that the above discussion to retrofit additional security involve changes to existing or developing specifications. As the APIs studied are relatively nascent, we are hopeful of a positive response from the community. In the present scenario, without modification, users of these APIs can use JavaScript analysis techniques to detect and eliminate such attacks during testing [67, 78]. Analysis systems similar to ours can be extended to taint data from `postMessage`, `localStorage` and `webDatabase`, ensuring that no tainted data flows to critical code evaluation constructs without sufficient validation. We have had some success in the past with such an approach [68, 67].

3.5 Related Work

Browser based abstractions and new browser features have been a topic of discussion in recent literature. Namely, the `postMessage` API was studied by Barth et al. as a browser based primitive capable of sandboxing mashup communications[26]. The authors used `postMessage` for frame communication and emphasize its utility as a primitive. In contrast, we focus our discussion on the difficulties in verifying an origin of a message.

A survey of the possible dangers of browser based storage was presented by Michael Sutton at Blackhat 2009 and discussed briefly in the book AJAX Security[76, 48]. These works differ from ours because we focus on showing that persistent browser based storage is a vector for stealthy, persistent attacks. We also show that robust, popular applications are already using this technology, despite the nascence of the API. Additionally, we propose solutions to detect and mitigate the threat of browser based storage.

A similar, but potentially more serious attack is persistent cross channel scripting(XCS), introduced by Bojinov et al., is an attack where a non-web channel is used to inject a script into web content running in a different security context[28]. They study devices like network attached storage and enterprise level lights management systems, which often have web management software built in, and show that they are susceptible to persistent injected content.

Automatically reverse engineering protocols was addressed by Caballero et al. with their prototype tool Polyglot[30]. However, they focus their efforts on using dynamic analysis to reverse engineer network protocols. An earlier work, Athena, aims to prove the correctness of security protocols[74]. In contrast, our efforts focused on manual protocol verification using tools such as Kudzu to analyze the JavaScript application.

Finally, a variety of tools have been developed for analyzing JavaScript for security vulnerabilities. Kudzu is a mixed concrete and symbolic execution engine which does information flow analysis of JavaScript programs in order to find potential vulnerabilities[67]. Another tool, Gatekeeper, takes a mostly static approach for analyzing JavaScript applications, and constructs multiple information flow policies for evaluating JavaScript[43]. In contrast, their work focuses on finding flows that exhibit a general policy violation in the code (such as untrusted data being written to the `GlobalStore` or untrusted data flowing to `document.write`).

3.6 Conclusion

New primitives, especially for browser-side functionality, are being designed and proposed at a rapid pace to facilitate the demand for interactivity while enabling security. However, a recurring problem in these designs is that these abstractions were not designed with the economy of liabilities principle in mind, i.e., they rely significantly on the developers to ensure security. In the case of `postMessage`, developers must check the origin of the message being received in order to ensure authenticity, however, nothing prevents a developer from omitting these checks. In the case of `localStorage`, developers are required to sanitize their data before or after it is inserted or read from the database. We noted that some applications performed sanitization and others did not. However, we cannot conclusively say that those who omitted checks did so without knowledge of the impact of their decisions— they may be confident in their platform's ability to perform sanitization, thereby omitting database level sanitization[85].

In this chapter, we found vulnerabilities based on usage of two recent client-side abstractions: `postMessage`, a cross-domain communication construct and client-side persistent storage (HTML5 and Google Gears). In the case of `postMessage`, we reverse engineered the client-side protocols and systematically extracted the security-relevant checks in the code to find new vulnerabilities in them. In the case of client-side storage, we found that applications do not sanitize database outputs, which can lead to a stealthy, persistent, client-side XSS attack. We found bugs in several prominent web applications including Gmail and Google Buzz and uncovered severe new attacks in major client-side protocols like Facebook Connect and Google Friend Connect.

We hope our study encourages future primitives to be designed with the economy of liabilities principle in mind. We offer some enhancements to existing to the current APIs to shift the burden of verifying and ensuring security properties from the developer to the browser. And, we encourage developers to scrutinize their applications for similar problems using automated techniques.

3.7 Acknowledgments

The Emperor's New APIs: On the (In)Secure Usage of New Client-side Primitives originally appeared at **W2SP 2010**. It is an original work by Steve Hanna, Eui Chul Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, Dawn Song. We thank Chris Grier, Adam Barth, Adrian Mettler, Adrienne Felt, Jon Paek and Collin Jackson for helpful feedback on the chapter and suggestions for improvements on the work.

Chapter 4

Take Two Software Updates and See Me in the Morning: The Case for Software Security Evaluations of Medical Devices

4.1 Introduction

Medical devices used for critical care are becoming increasingly reliant on software; however, little is understood about the security threats facing medical devices and their software. To investigate this open question, we analyze the security of software that controls a modern Automated External Defibrillator (AED) used for treating cardiac arrhythmias. We identify several software security vulnerabilities and discuss key insights and open challenges in improving software-controlled medical devices to be resistant to malware[1].

4.2 AED Overview

In this chapter, we introduce automated external defibrillators and discuss why they are a quintessential software-based medical device to investigate. We point out that AEDs are significantly different from implantable cardiac defibrillators both physically in terms of their hardware, software and connectivity, and in terms of risk. We discuss AEDs in use and introduce the Cardiac Science G3 Plus — the subject of our evaluation.

An AED allows a person with limited medical training to defibrillate a patient who experiences certain kinds of cardiac arrhythmias that, untreated, can lead to cardiac arrest. The device is typically attached to the chest of the individual using two pads, and it delivers an electric shock to reestablish a normal heart rhythm. The conditions that can be treated through defibrillation, for example cardiac arrhythmias of ventricular fibrillation and ventricular tachycardia, generally require prompt attention in order to prevent severe brain damage

[1] In this chapter, I was responsible for jointly reverse engineering and finding vulnerabilities, guiding exploit development and jointly creating the proposals for securing medical devices.

or death. Studies have shown that the chances of survival improve if an individual receives defibrillation within 3-5 minutes of collapse [3]. For this reason, AEDs are widely available in airports, community centers, schools, government buildings and other public locations. As of 2010, there are about 280,000 AEDs worldwide [35].

The FDA has received more than 28,000 adverse event reports for external defibrillators between January 2005 and May 2010, including malfunctions, patient injuries and deaths. As a result, the FDA has issued 68 recalls, 17 alone in 2009. In U.S. hospitals, more than 50,000 AED units were sold between 2003–2008, with a projected annual sales growth of 9% to 12% over the next 5 years [31]. In 2005, over 192,400 AED units were sold in the U.S. marketplace [71]. 16.2% of AED advisories were confirmed to be as a result of software-related issues (6 advisories affecting 12,311 individual devices). Only one category had a higher incidence (electrical issues at 21.6% of the causes of a recall) [71]. Given the importance of these devices in saving lives, the FDA began an initiative to promote the development of safer external defibrillators in November of 2010 [8].

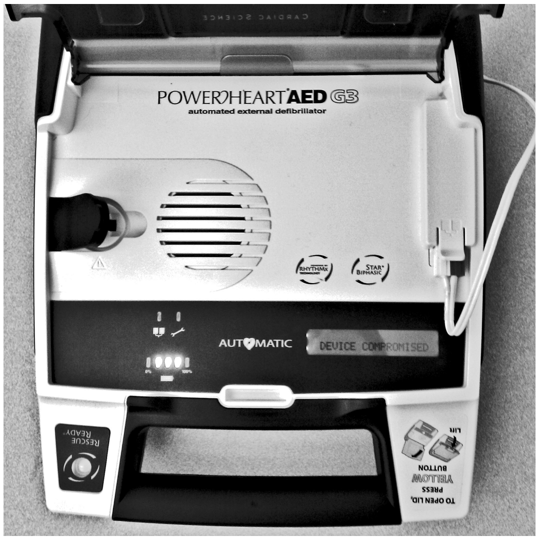


Figure 4.1: The Cardiac Science G3 Plus exploited to install our custom firmware. The AED displays DEVICE COMPROMISED.

In this chapter, we study one particular model of AED, the Cardiac Science G3 Plus with model number 9390A-501. We decided to examine an Automated External Defibrillator (AED) because of its reliance on software updates to address an FDA recall. Depending on how the device is configured, the shock administered can be 150-300 Joules, which can be administered multiple times on one battery before the device requires a recharge. The G3 Plus is pictured in Figure 4.1.

4.3 Case Study

In this chapter, we analyze the Cardiac Science G3 Plus Automated External Defibrillator. We obtained the device new from an online vendor and the firmware update and software suite from the Cardiac Science website. In order to analyze the device, we used IDA Pro 5.6, an inexpensive, commercially available software package, to statically reverse engineer the device's update and diagnostic software, communication protocol, and firmware [13]. Our analysis was conducted using COTS hardware and software, and took upwards of 100 hours. During our analysis, we discovered that **4** vulnerabilities while reversing the device and its software. We verified our analysis and show the gravity of software exploits on medical devices by executing concrete attacks and demonstrating their potential impact.

4.3.1 Analyzing the G3 Plus

Performing a security analysis of the G3 Plus necessitated reverse engineering device operation and controlling software because it is a closed-platform device, with proprietary hardware, firmware and software.

As a first step in the investigation, we had to determine which components were responsible for the AED's functionality. In order to do this, we opened the G3 Plus, and we noted the chip model numbers, which were later cross-referenced with known components. The devices of interest were an Actel FPGA, an OKI LSI Audio Playback FIFO chip, and an Intel 16-bit High Integration Embedded Processor. The playback IC is used for playing audio prompts to the user to guide resuscitation. It is also used to play diagnostic messages. We speculate that the FPGA is used to detect heart rhythm and assess a patient's cardiac health.

We discovered that the AED's CPU implements a 16-bit x86 ISA by examining the internals of the AED with CPU processor manuals. From this, we applied static reverse engineering techniques to the *firmware*.

The device comes with three software packages: *MDLink*, *RescueLink* and *AEDUpdate*, responsible for programming device parameters (like shock value), collecting post-cardiac arrest reports, and updating the AED, respectively. We focus our analysis on the device firmware (252KB), *AEDUpdate* (8MB) and *MDLink* (680k) because of their potential impact on correct device operation; they are responsible for communicating with the host computer, updating firmware, and programming device parameters.

We also used the device's manual to determine operational parameters and constants, which we used as clues to firmware functionality.

4.3.1.1 Preliminary Analysis

We use the functional descriptions of the software to guide our analysis of the G3 Plus. After providing a basis for our analysis, we delve into the details of the device and described vulnerabilities contained therein.

Isolation and Architecture. We discovered that the G3 Plus platform is centrally controlled by an Intel 16-bit High Integration Embedded Processor, which implements a 16-bit x86 instruction set. The architecture supports a single execution thread in a shared memory model without any signs of virtual memory, segmentation or any other memory isolation or virtualization techniques.

Firmware. We noted, while exploring the *AEDUpdate* software, that the device's firmware is updated by sending the new firmware over the COM port. Using this observation, we were able to reverse engineer the code to determine that a local copy of the firmware existed hidden in the Cardiac Science program directory. When an update is initiated, the software determines the AED connected, and selects the corresponding firmware. Having both the firmware and copies of the medical device software, we are able to delve deeper into exploration by applying binary analysis techniques.

Language, Libraries, and Execution Safety. We began studying the firmware by executing a preliminary analysis with IDA. We set out to determine the compiler used and libraries utilized. As an initial step, we had to manually discover the firmware entry point. We used the Intel processor manual to discover that the entry point[2] is fixed at address *0xFFFF0*. Using this, we applied IDA FLIRT signatures and control flow analysis.

The FLIRT analysis aided us in determining if a known library was compiled into the firmware. As a preliminary step, this serves to partition firmware code into library functions and device code, both offering us insight into the library and its intended function. This also allows us to focus our analysis on the portion of the firmware responsible for controlling the defibrillator, and then use the FLIRT-identified library functions as clues to infer firmware function behavior from context.

We determined the compiler by applying each of IDA's FLIRT signatures applicable to 16-bit x86 code. One such signature, "MSVC v1.0/v1.5 DOS runtime," resulted in over 200 sequentially-located functions being recognized as library functions. The sequential locality matched our expectations of how statically-linked libraries are typically compiled into binaries. A manual analysis confirmed that these functions behaved in the ways that their FLIRT-identified library names suggested.

Further manual inspection of the code suggested that the firmware was unoptimized, did not employ any sort of binary packer, and did not employ obfuscation techniques[3]. Based on the libraries compiled into the firmware, the compiler used and manual confirmation, we determined that the firmware was written in C and 16-bit x86 assembly, and the majority of functions conformed to the `cdecl` calling convention.

Attack Vectors. As discussed previously, the device is equipped with a serial port for communication with the host computer. We discovered that the serial port is used by *AEDUpdate*, *RescueLink*, and *MDLink* for device updates, post-defibrillation reports, and

[2] In the Intel manual, this is known as the Reset Vector.

[3] Although, some optimizations are inherently invisible and difficult or impossible to recognize: function inlining and loop unrolling, for example.

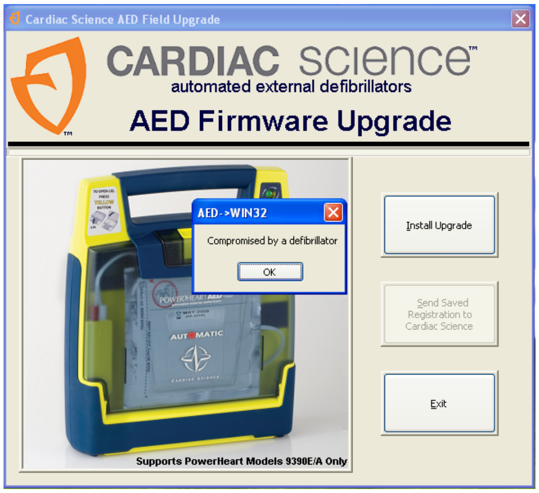
device parameters, respectively.

4.3.2 Vulnerabilities in the Cardiac Science G3 AED

Below we detail the key findings of our investigation into the security of the AED and its software. First, we discuss the bugs and vulnerabilities we found in the *MDLink* and *AEDUpdate* software. Following that, we discuss how we can arbitrarily change the G3 Plus's firmware.

***Vulnerability 1**: AEDUpdate integer/buffer overflow.* Initially, *AEDUpdate* sends a packet over the COM port to the AED and then receives a response. In order to verify the vulnerability, we spoofed packets coming from the COM port such that we were able to control the data that *AEDUpdate* received from the device, and thereby triggered the vulnerability and achieved arbitrary code execution.

The *AEDUpdate* program counts the number of '0'-characters in the response and then performs a `memcpy` based on the number of zeros found. Specifically, the program expects to find no more than 8 zeros, so it uses the expression `8-num_zeros` to calculate the size of the buffer to copy. However, if there are more than 8 zeros, the *size* parameter underflows and results in attempting to copy around 4.3 billion bytes of data into a 16-byte buffer, causing the program to throw an exception. In order to exploit this vulnerability, we need to overwrite the most recently registered exception handler with our own and cause this exception to occur before the corrupted return address is checked, which allows us to redirect program flow into arbitrary code.


Cardiac Science AED Field Upgrade
CARDIAC science™
automated external defibrillators
AED Firmware Upgrade
Install Upgrade
Send Saved Registration to Cardiac Science
Exit
Supports PowerHeart Models 9390E/A Only
AUTOMATIC
OK

AED->WIN32

Compromised by a defibrillator

Figure 4.2: *AEDUpdate* buffer overflow. Executed code includes a message box showing the potential flow of the vulnerability from the AED (if the firmware were replaced) to the software.

In Windows XP SP2, nearly every module linked into a binary uses Safe Structured Exception Handling (SSEH). Meaning, the exception handler must be registered with the operating system, and if during execution it is discovered that it is not registered, the process is terminated by the OS [18]. `oledlg.dll` was the only DLL imported into the binary that did not employ SSEH. Using this, we *overwrote* the exception handler on the stack with a subverted one, forcing the program to execute our payload when an exception was thrown[4]. Figure 4.2 shows *AEDUpdate* being exploited.

***Vulnerability 2**: Weak password authentication scheme.* The *MDLink* software has individual user profiles protected by a password. However, the password file is stored on the local hard drive, and anyone with privileges can simply delete it in order to circumvent the protection mechanisms or circumvent it completely by installing the software fresh on a machine that does not have access restrictions. Furthermore, after we reverse engineered the *MDLink* authentication mechanism, we discovered that the password was obfuscated using a simple `XOR` scheme. From there, we extracted the scheme and wrote a small utility to arbitrarily change or recover a user's password.

***Vulnerability 3**: Credentials stored in plaintext.* While examining *AEDUpdate*, we came across functionality that seemed to upload a file to a remote FTP server. Further investigation revealed that we had found that the *AEDUpdate* stores the address of the FTP server, username, and password in *plain text* within the update software. When warranted, either due to a failed upgrade or other diagnostic problem, the diagnostic file is sent to Cardiac Science. This private user information is collected when a firmware upgrade is started by *AEDUpdate*. and includes: contact information, serial number, and the firmware version — all potentially available to anyone who accesses the login credentials.

***Vulnerability 4**: Improper use of weak CRC as digital signature.* We discovered that the update software begins the update process by sending a Cyclic Redundancy Check (CRC) value of the new firmware. Once transferred, the device calculates the CRC of the received firmware and allows the current firmware to be replaced only if the values match. In order to install custom firmware, we populated its CRC verification table with new values calculated over the entirety of our custom image. To obtain the precise integrity check employed by the AED, we extracted the code responsible for CRC calculation and verification from *AEDUpdate*, which we used to generate a new CRC to convince the AED that our image was a legitimate one.

This vulnerability allows malicious modifications that could lead to device failure, patient harm, and financial burden. These attacks include: (1) Disabling or falsifying integrity checking so that when a component fails, the device will pass verification; (2) Setting arbitrary shock protocols and shock strength; (3) Failing to administer shocks despite device appearing to operate as normal; (4) Bricking the device; (5) Destroying the AED's audit trail; (6) Resource exhaustion and DoS by exhausting the devices battery and (7) Executing a conditional payload, depending on the date or time.

[4] For instance, by corrupting a function pointer on the stack.

4.3.3 Impact

Besides the direct security implications of the vulnerabilities discovered, perhaps less obviously, the resulting vulnerabilities, namely *Vulnerability 1* and *Vulnerability 4* can be combined to create a hybrid piece of malware capable of infecting both AEDs and PCs. Consider the scenario where an attacker replaces the firmware of an AED with custom malware designed to exploit the *AEDUpdate* buffer overflow. This would allow the AED to execute arbitrary code. An attacker could create a worm that infects the host computer and subsequently infect other AEDs connected to the host. This type of attack could have the purpose of damaging other equipment within an organization or obtaining private data stored in systems other than the affected medical devices. Threats like these punctuate the necessity of designing medical devices with security in mind.

4.4 Automatic Vulnerability Discovery in Medical Device Software

Dynamic analysis and symbolic execution are a viable way to systematically explore medical device software for bugs and vulnerabilities. We obtained BitFuzz, which is based on the BitBlaze framework, an open-source framework for trace-based dynamic symbolic execution of COTS software[73].

BitFuzz is particularly suited for confirming the results of a manual reverse engineering effort (see Chapter 4.4); and it can be used to uncover other potential software vulnerabilities. BitBlaze is comprised of three main components, *TEMU*, *Vine* and *BitFuzz*. *TEMU* is a full system emulator with dynamic taint tracking, which is used for recording execution traces. This is used in conjunction with *Vine*, which converts the trace based format into a side-effect free, RISC-like instruction set that captures the semantics of the original operation. Finally, *BitFuzz* is used for generating inputs which cause a bug to be triggered. It drives the guided exploration of the program by analyzing the intermediate representation of each execution trace and generates precise logical formula leading to a particular branch condition, which can be extracted and solved.[5] Using the solution to a branch condition, BitFuzz negates conditions and explores previously unreachable branches in the program.

Using BitFuzz for Software Flaw Confirmation. As we alluded to previously, BitFuzz combined with insight gained by reverse engineering provides the necessary tools to apply binary analysis techniques to automatically investigate areas of interest. We detail how we used BitFuzz to confirm our suspicions of software flaws discovered when reverse engineering the G3 Plus update process.

We were specifically interested in discovering how the program communicated with the AED as well as the format of the protocol in hopes of discovering how to replace the device firmware with our own. We investigated each function that read or wrote to the

[5] This assumes that the decision procedure determines that the logical formula is satisfiable and that the logical formula falls within a theory of arrays and bit vectors.

serial port and annotated it with suspected functionality. We specifically explore each use of `ReadFile` Windows API function because it is responsible for reading the input stream from the serial port. In the process of determining how the program updates the AED firmware, we noticed that when the firmware is about to be updated, an initial handshake takes place between the host and the AED, followed by sending the configuration of the connected AED to the host computer.

We found that the configuration protocol had 9 packet payloads with 128 bytes in each packet. While investigating the initial handshake (first 3 packets), we found a suspected flaw in which *AEDUpdate* seemed to make incorrect assumptions about the format of the reply packet. That is, we found a `memcpy` in the packet parsing code whose *size* parameter was calculated by subtracting the number of '0' characters in the header from 8. We were unable to find code that ensured that the *size* parameter was *not* negative before invoking `memcpy`. Further investigation suggested that we had found an integer buffer overflow; however, in order to verify our suspicions, we had to ensure that the integrity of the packet was not verified elsewhere in the program. In order to side-step the laborious task of completely reverse engineering the *AEDUpdate* binary, we used BitFuzz to determine that we had in fact found a bug in the software. Below, we discuss the experimental setup.

Experimental Setup. We ran BitFuzz in Linux on a 2.4Ghz Quad Core Intel CPU with 4GB of RAM. We marked `ReadFile` as a taint source and configured BitFuzz to repeatedly force *AEDUpdate* to execute the configuration handshake. BitFuzz confirmed that a bug existed in *AEDUpdate*. On the 94th input stream generated, the program crashed. This occurred after running BitFuzz 77 minutes and 20 seconds.

4.5 Improving Software-based Medical Device Security

In order to improve upon the security of medical devices, we discuss four key principles which, based on our experiences, mitigate the attacks outlined in this chapter against the G3 Plus. Most importantly, we discuss future areas of exploration along with defenses and stress that solutions must be consider along with the potential risk to the patient.

4.5.1 Cryptographically secure device updates are needed to ensure firmware integrity

An end-to-end secure device update prevents attackers from replacing medical device firmware. Recall that the G3 Plus' update mechanism relies on a cyclic redundancy check to detect errors in transmission of data to the device but lacks a mechanism to guarantee firmware authenticity. We propose that in light of this, devices like the AED with on-device firmware should be designed to only accept signed firmware.

However, the solution to this in not obvious. In addition to having strict power constraints, a complete solution to the problem of providing secure software updates for embedded devices has not yet been found. Attempts have been made to extend a Trusted Computing

Platform for software-supported and hardware-supported updates [52]. However, solutions applying to software medical devices have not been explored.

4.5.2 Device telemetry verified for integrity and authenticity

Software or devices that rely on medical device telemetry should treat input data as adversarial and design the device's software to handle a wide range of malformed or malicious inputs. As we saw with the G3 Plus, the update software did not correctly handle the case where a packet was malformed. Accurate modeling of device protocols and testing are the obvious choices to ensure correct operation. Furthermore, greater assurance of correct device operation is attainable by implicitly not trusting telemetry and programmatically employing rigorous sanity checks on inter-device communication. Good programming practices coupled with rigorous static analysis checking for security vulnerabilities mitigates many threats, and in the case of the G3 Plus, would have prevented the overflow that lead to arbitrary code execution [64].

4.5.3 Passwords should be cryptographically secure and easily managed

The two password mechanisms we found in the program relied on obscuring the location or text of the password, which made them easily recoverable. In medical devices, where passwords potentially provide access to private date or life-critical functionality, they must be securely protected. However, traditional protection mechanisms are muddied by the added complexity of managing passwords in an environment where potentially many people need to access the same device or software functionality. Furthermore, password revocation and access control add to the intricacy of a secure design.

4.5.4 Defenses and updates must be weighed with their risk to the patient

Mechanisms that ensure and maintain device integrity must be weighed with their potential risk to the patient. With respect to updates: What is the purpose of the update? How might it change device operation? Is the update crucial to patient safety? In the case of defending against modification, if a device's firmware has been modified, it may be necessary to identify with certainty the severity of the potential integrity flaw. In other words, the process of verifying that a device has not been tampered with should not make it easier for a malicious party to launch a denial of service attack on a device with critical functionality.

Most notably, we point out the general principal that medical devices, in most cases, need to *fail open*. This is particularly true when emergency access to medical devices is needed. For instance, if a patient with an implantable defibrillator collapses, the treating doctor would need to query the device for information in order to treat the patient. Simply denying access to the doctor is unacceptable, as it puts security ahead of accessibility which is in direct contention with the usability of the device [45].

This presents a major design challenge for device developers because they are faced with the difficult problem of providing security while judiciously restricting accessibility.

4.6 Conclusion

Software security remains an afterthought for medical device design. Our case study of a popular Automated External Defibrillator (AED) identified security flaws in both the embedded software and the COTS software update mechanism. For future medical devices to remain safe and effective in the presence of fast wireless communication and dense Internet connectivity, medical device software designers must learn how to improve their software security and explore solutions to withstand malware while confronting the unique challenges in the domain.

4.7 Acknowledgments

Take two software updates and see me in the morning: The Case for Software Security Evaluations of Medical Devices originally appeared at **HealthSec 2011**. This chapter is an original work by Steve Hanna, Rolf Rolles, Andrés Molina-Markham, Pongsin Poosankam, Kevin Fu and Dawn Song. A special thanks to Rolf Rolles for his reverse engineering expertise and guidance, this work would not have been possible without him.

Chapter 5

Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications

5.1 Introduction

Mobile application markets such as the Android Marketplace provide a centralized showcase of applications that end users can purchase or download for free onto their mobile phones. Despite the influx of applications to the markets, applications are cursorily reviewed by marketplace maintainers due to the vast number of submissions. User policing and reporting is the primary method to detect misbehaving applications. This reactive approach to application security, especially when programs can contain bugs, malware, or pirated (inauthentic) code, puts too much responsibility on the end users. In light of this, we propose Juxtapp, a scalable infrastructure for code similarity analysis among Android applications. Juxtapp provides a key solution to a number of problems in Android security, including determining if apps contain copies of buggy code, have significant code reuse that indicates piracy, or are instances of known malware. We evaluate our system using more than 58,000 Android applications and demonstrate that our system scales well and is effective. Our results show that Juxtapp is able to detect: 1) *463* applications with confirmed buggy code reuse that can lead to serious vulnerabilities in real-world apps, 2) *34* instances of known malware and variants (*13* distinct variants of the GoldDream malware), and 3) pirated variants of a popular paid game.

Contributions[1]**.**

- *Vulnerable Code Reuse.* We show that applications widely use significant portions

[1] **In this Chapter, I was responsible for development of the single user version of the tools and methodology(excluding clustering) and the evaluations (excluding performance evaluations).**

of the Google In-App Billing and License Verification example code, leaving them susceptible to vulnerabilities.

- *Instances of Known Malware*. We find **34** instances of malware in Android markets, **13** of which are distinct, previously unknown variants that have been repackaged with innocuous-looking applications.
- *Piracy*. We identify pirated applications in third party markets and show that Juxtapp can detect pirated applications that are obfuscated and with significant code variation from the original application.

5.2 Problem Definition

In this chapter we consider the problem of automatically finding similarity among Android applications with the goal of detecting known buggy code patterns and vulnerabilities, repackaged and pirated applications, and known malware in Android markets. Detecting code reuse in Android applications offers a first chance in detecting applications that may negatively impact the user's security and experience or defraud developers of revenue. We develop Juxtapp, an architecture that automatically examines code containment in Android applications. We define code containment to be a measure of the relative amount of code in common between two Android applications. Using this, we examine a variety of Android market applications for instances of vulnerable code, known malware, and piracy.

Buggy and Vulnerable Code Reuse. Previous manual investigations into developer errors[32, 37] in Android applications have indicated that developers often copy and paste code as well as reuse sample code obtained from Android-specific developer websites without modification. Using application similarity, we can examine the Android Market to see if they contain known buggy or vulnerable pieces of code.

Known Malware. The incidence of malware in Android marketplaces has been rising rapidly. In January 2011, 80 applications were known to be infected with malware, as opposed to June 2011, when the incidence rate had risen to over 400 instances of malicious applications [15]. Malware authors often repackage legitimate applications with a malicious payload in order to entice users to download an infected application.

Piracy and Application Repackaging. Popular Android applications and games are commonly repackaged with modified code in order to evade copyrights protection and to generate revenue for the pirate [6]. By comparing applications from the official Android market to third party markets we show that we can detect instances of piracy.

Scope. We restrict ourselves to the Android application domain, excluding obfuscation in the form of functional code transformation. For instance, we are able to detect two instances of similar obfuscated code, but we restrict ourselves to this domain and do not consider the problem of matching code which has been transformed to be functionally equivalent.

5.2.1 Goals and Challenges

Juxtapp has a variety of challenges which must be met in order to detect code reuse in Android applications. Some specific goals of our platform are to:

Automatically analyze code similarity in Android applications. As of November 2011, the Android market had over 310,000 applications[16]. The rapid growth of market applications and increase in the number of pirated and malicious applications underlines the need for a way to rapidly and automatically analyze applications.

Scale to a large number of applications. Android markets have hundreds of thousands of applications with new applications being added all the time. Our architecture must be able to scale in order to detect similarity across a wide range of applications, including the ability to *incrementally* update our application repository in an efficient manner.

Accurately and efficiently represent the applications under analysis. In handling hundreds of thousands of applications, Juxtapp must be able to accurately represent and quickly determine code similarity among applications. There is an implicit trade-off between the accuracy of the analysis and the amount of space it takes to represent an application under analysis.

5.2.1.1 Android Specific

In addition to general challenges, there are a number of domain specific considerations when computing the similarity of Android applications.

Java Source Code Unavailable. For most applications on the Android Markets, source code is not available. Android applications are compiled from Java to Dalvik bytecode (known as the DEX format)—the bytecode for the Dalvik VM[5]. This compiled code and application resources are packaged as an APK. The DEX format fully describes the application and retains class structure, function information, etc.

Multiple Entry Points. Unlike traditional desktop programs, Android applications have multiple potential entry points. Android applications are broken up into components and these components can each have their own entry points.

Obfuscation. Android developers are encouraged to obfuscate their code using Proguard[17]. Proguard attempts to remove unused code and renames classes, methods, and fields with obscured names to make reverse engineering of Android applications more difficult. However, this process is deterministic so two identical applications will be transformed in the same way.

Therefore, any type of program analysis must take these challenges into consideration. And indeed, the domain specific challenges can be used to impose structure on the applications so that feature hashing and clustering are more amenable in the Android application domain.

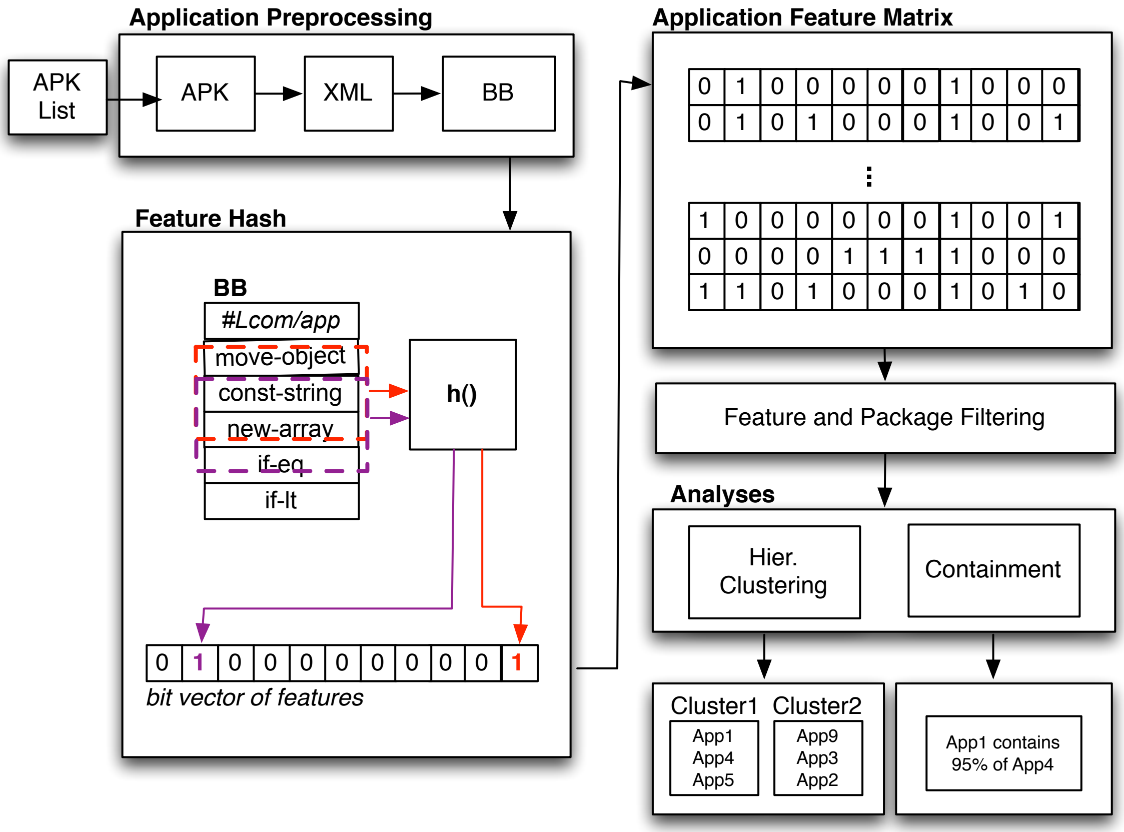


Figure 5.1: The Juxtapp Workflow.

5.3 Background

Like static code reuse detection proposed in [82, 58], we use k-gram features of code sequence to represent applications. However, k-grams extracted from code sequence usually results in an enormous feature space (e.g., 2^{128} features in [82, 58]), preventing efficient feature storage and similarity comparison even for a moderate number of applications. To analyze large volumes of mobile applications, we need an efficient feature representation of the applications and a fast way to compare features between them.

Feature Hashing. The main technique we use is feature hashing. Feature hashing is a popular and powerful technique for reducing the dimensionality of the data being analyzed [54, 72]. Using a single hash function, it compresses the original large data space into a smaller, randomized feature space, in which feature hashing, representation, and pairwise comparison are all efficient. This efficiency comes at the cost of potential collisions while hashing. However, theoretical and experimental results from the machine learning community show that pairwise similarity maintains high accuracy, thus algorithms built on top like hierarchical clustering, will be close to exact [54, 72]. Feature hashing was recently introduced into the security community for malware analysis [50].

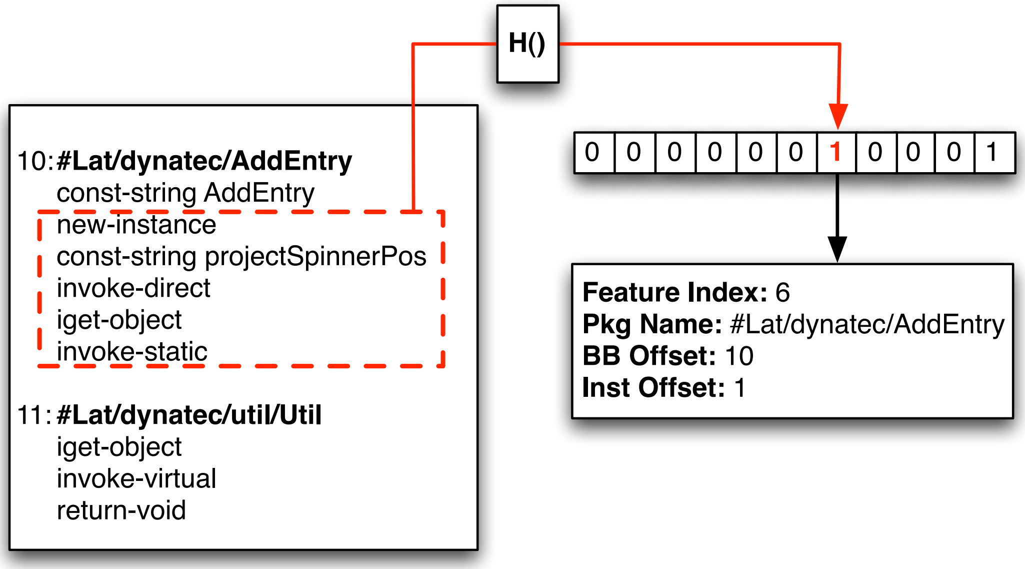


Figure 5.2: Example outcome of feature hashing a basic block found in an application.

The resulting representation of an application can be encoded into a succinct bitvector which represents the features present in the data. As always, choosing a good hash function and a bitvector representation of prime length is essential to minimize the number of collisions in the vector.

Similarity. We determine the similarity of two applications by the similarity between their feature sets. We use the Jaccard similarity metric defined as $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, where A and B are two k-gram feature sets of two applications, respectively. Because we hash k-gram sets into boolean vectors, with each entry indicating presence or absence of a feature, as opposed to a set of items, we can approximate this quantity much more efficiently using bit-wise operations: $J(\hat{A},\hat{B}) = \frac{|\hat{A} \wedge \hat{B}|}{|\hat{A} \vee \hat{B}|}$, where $\hat{A}$ and $\hat{B}$ are bitvector representations of k-gram sets A and B, respectively. As shown in [50], as long as the size of the bitvector is large enough, $J(\hat{A},\hat{B})$ is very close to J(A,B), the similarity between two applications represented by the k-gram feature sets. The Jaccard distance $D(A,B)$, which measures *dissimilarity* between two feature vectors, is obtained by subtracting the Jaccard similarity from one: $D(\hat{A},\hat{B}) = 1 - J(\hat{A},\hat{B})$. Both Jaccard similarity and distance have values in the range $[0,1]$.

5.4 Our Approach

As shown in Figure 5.1, our approach, Juxtapp, involves the following steps for analyzing Android applications: 1) application preprocessing, 2) feature extraction, and 3) clustering and containment analyses.

5.4.1 Application Preprocessing

We preprocess each application in order to extract the DEX file, which represents the compiled application code. In our approach, the original Java source code is *not required* because the DEX format fully describes the application and retains class structure, function information, etc.

For each application we convert its DEX file into a complete XML representation of the Dalvik program, including program structure. From this, we extract each basic block and label it according to which package it came from within the application. We process each basic block and only retain the opcodes while discarding most operands. The exception to this is for opcodes storing constant data, such as the `const-string` opcode, which becomes a concatenation of the opcode along with the value it references.

The intuition behind this is that many Java applications contain boiler plate code that will appear in many applications when only opcodes are considered. Furthermore, including constants makes the feature hashing (discussed below) more fine-grained and more restrictive about matching. This is especially important because many applications use Java reflections to access functionality, with the only difference between programs being the string arguments passed to the Reflections API.

5.4.2 Feature Extraction

We use k-grams of opcodes and feature hashing to extract features from applications. We use the `djb2` hash function which is known to have an excellent distribution[12]. As shown in the *Feature Hash* box in Figure 5.1, for each application's basic block representation of the original XML file, we extract each k-gram using a moving window of size k, and hash it using `djb2`. k-grams across basic blocks are ignored. For each hashed value, we set the corresponding bit in the bitvector of the application, indicating existence of the k-gram. Along with this information, we efficiently store the package name from which the basic block originated, the basic block offset within the basic block file, and the k-gram offset. This allows us to recover how and why our architecture determined that applications are similar and serves as a way to verify matching applications.

In order to feature hash, we have two parameters to determine, namely: length of k-gram k and bitvector size m. In Chapter 5.5, we show an experimental evaluation of several values of k and m in order to determine optimal values of these parameters over our dataset.

Choosing k. k is a parameter which determines the number of dimensions of the underlying feature space for representing the Android applications, and it bounds the number of features that can be extracted for each application. k is a crucial parameter for detecting similarity. If k is too small (e.g., $k = 1$), there will be a small number of unique features from all applications, resulting in an oversimplified, low-dimensional representation of the applications. In this representation, overmatching between applications can occur, and many applications would be falsely classified as being similar applications would be falsely similar. On the other hand, if k is too large (e.g., bigger than the size of most basic blocks), even small code changes might result in large changes in the feature representation, preventing us from obtaining a meaningful and robust comparisons between applications. In general, a reasonable k should have a small value, at which further increase in value would cause insignificant increase in the quality of the pairwise similarity comparison. As shown in Chapter 5.5, we evaluate different k values, and choose $k = 5$, at which its marginal impact on similarity accuracy is around 0.01.

Choosing Bitvector Size. The bitvector size m strikes the tradeoff between (similarity) computation efficiency and approximation error of the bitvector representation of the k-gram features.

Ideally, we want size m to be large enough so that few collisions would happen when we feature hash k-grams into bitvectors; practically, we want size m to be small so that we can efficiently compute pairwise similarity among hundreds of thousands of applications. The larger the bitvector size m, the more accurately a bitvector represents an application, but at the cost of more time required to compute the pairwise similarity among all applications.

As shown in [50], as long as $m \gg N$, which is the number of k-grams extracted from an application, the Jaccard similarity between two bitvectors very closely approximates computing the set intersection between two k-gram feature sets. That is, as long as m is large enough, Jaccard similarity is nearly an exact representation in practice. Based on this principle, we use a data-driven approach in our experiments in Chapter 5.5, in order to determine a bitvector size which is large enough to represent the feature space in question.

5.4.3 Analysis of Feature Hashing Results

A variety of data analyses can be performed on the feature representation of the applications. In this paper, we primarily focus on similarity, containment and clustering analysis, which help us to filter out vast amounts of uninteresting instances and pare down a small set of interesting candidates for further analysis.

5.4.3.1 Code Containment Comparison

Containment analysis is a useful tool for paring down application candidates that potentially have copied code, pirating, and malware contamination. We define the containee A to be the application being examined for similarity and the container (or carrier [2]) B to be the application which houses the packages and associated features that we test for existence inside the containee. We define a metric that gives the percentage of containment by considering the number of features common in both applications, divided by the number of bits in the containee application. Formally, containment is defined as: $C(\hat{A}|\hat{B}) = \frac{|\hat{A} \wedge \hat{B}|}{|\hat{A}|}$. Written in this form, this containment is defined as the percentage of features in application A that exist within application B.

5.4.3.2 Clustering

To find inherent patterns among Android applications, we use agglomerative hierarchical clustering[34] on the feature bitvector representation of each application in order to group similar applications together. The basic idea is that the collection of feature bitvectors represents the applications in a high-dimensional space with a well-defined distance metric, the Jaccard distance. Using this distance metric, we can group bitvectors that are close-by and, thus, we are able to group similar applications.

Hierarchical clustering produces clusters without having to specify the number of clusters in advance. The input to the clustering algorithm is a threshold t (e.g., 90%) and a list of Jaccard similarity values between each pair of applications. The output is a clustering S for the applications, in which all applications in a cluster are with similarity s greater than or equal to $t : s \geq t$. The threshold t is set by the desired precision tradeoff between the number of applications in the clusters and the "closeness" of applications within a given cluster. While a smaller t puts more applications into a few large clusters, a larger t discovers specific variants of application families (e.g., similar applications developed by the same authors).

Hierarchical clustering begins with one application in its own cluster; then it selects the closest pair and merges them into a common cluster. The cluster comparing and merging process continues until there is no pair whose similarity exceeds the input threshold t.

For clusters with multiple applications, we use single-linkage to define the similarity between them, i.e., the similarity between cluster S_a and cluster S_b is the maximum similarity between all possible pairs, i.e., $J(S_a, S_b) = max\{J(A, B)|A \in S_a, B \in S_b\}$.

5.4.4 Core Functionality and Result Refinement

Clustering can be a way to visualize the application topology in order to qualitatively understand how well applications are classified among a given cluster. Application similarity can be dominated by large similar libraries common to many applications (e.g. AdMob). In light of this, we develop the notion of *core functionality*, which seeks to capture in a coarse-grained manner how included libraries interact with the main application component.

[2] In the case of malware, a carrier is a more appropriate term because the innocuous application is modified in order to execute code outside of the intended functionality.

That is, the main package of the application is the core application component, and any direct edges from that component to other libraries makes that package a core application component.

Simply put, we examine each application and whether or not the core application component directly invokes an outside library. If the core application invokes a method in the library, then is a part of the application's functionality; otherwise, that code can be excluded from the analysis. For instance, AdMob, generally speaking, has no direct edges from or to its library code to the main application component, indicating it can be excluded in most applications [3].

We refer to the set of libraries excluded as an *exclusion list*. We point out that the exclusion is an over-approximation and aggressively excludes libraries. Because the exclusion list only looks at direct invocations to other packages and libraries, it will miss invocations due to dynamically called functionality, such as: Java reflections, dynamically registered event handlers, and other entry points implicitly defined by the Android Manifest.

In practice, we didn't find that the automatic generation of the exclusion list in the current implementation of this approach worked very well. In general, packages containing library code which many applications invoke, may only have a single piece of functionality used, but an inclusion of the entire library causes them to be falsely similar. This is because they actually use different functionality from within the same package. Future work will attempt to make this more fine-grained, at the package and method level.

Despite automatic generation of the exclusion list not being perfect, the automatic generation can be manually pruned or added to by an analyst. In practice, we have found analyst-driven modifications to the exclusion list works well, as we discuss in Chapter 5.5.

5.4.5 Implementation

The workflow of Juxtapp can be roughly broken up into the following stages: application preprocessing, feature hashing, clustering and containment analysis. Juxtapp consists of 6,400 lines of C++, 1,600 lines of Java, and 600 lines of scripts.

The first step in the process is converting the Android application file (APK) to a format which is usable by our architecture. Juxtapp processes the APK and outputs the file in an XML format with functions split into basic blocks, which is then converted to a basic block format, which has a label indicating the source package, class and method.

After preprocessing, the applications are feature hashed. Juxtapp processes the basic block file for each application and outputs a feature vector representing the application along with recovery information to verify matching portions of applications. That is, in addition to the features, we also store the package and class name, and the offset within the original file in order to verify matching potions of applications. Figure 5.2 shows an example basic block being feature hashed, along with the recovery information we store for each feature. For

[3] AdMob is invoked through the applications manifest or layout file, which we consider to not be a direct invocation to the application.

each program under analysis, the features calculated are stored as a sparse representation of the vector, while a table of each feature's offset within the original program along with the package and class from which it originated. The feature hash file stores the size of the vector, the number of entries that are set to one, and a list of features appearing in the program. The table file stores a feature number, the basic block offset within the file, the instruction offset within that basic block, and finally the class, package, and method name that the basic block was derived from.

After processing all of the applications' basic block files, Juxtapp calculates a pairwise distance matrix between all applications. This matrix is used for clustering and determining similarity among applications. We note that Juxtapp is capable of incremental analysis which allows additional applications to be considered without recalculating the entire dataset.

After the applications have been feature hashed, Juxtapp can perform other in-depth analysis. First, the applications under analysis can be clustered based on their computed distance matrix, which offers a topological view of the dataset, which can help an analyst narrow down interesting areas to investigate. For instance, after clustering, we combine author information with the applications in each cluster. This allows us to understand which applications came from which authors and helps us identify interesting candidates, i.e., those with conflicting authors and similar but modified application code.

Finally, Juxtapp computes containment between sets of applications. Given a set of feature hashed applications represented, the containment tool determines what features are common between applications and outputs the percentage of code in common, along with the ratio of the comparative sizes of the number of features. The ratio is important because of the asymmetric notion of containment which may indicate a high degree of similarity between two applications, but this is only a meaningful comparison if the applications are of the same relative size. For instance, a very small application may have many bits in the bit vector in common with a very large application, but this is due to the density of the larger application's bit vector and not necessarily a meaningful comparison. Simply put, a large application when compared to a small application may inadvertently have a large subset of the smaller applications features by virtue of the fact that a larger application will produce a dense feature vector. This ratio is used to remove false positives.

Distributed Analysis. We have both a single machine implementation of Juxtapp as well as a distributed implementation which uses Hadoop on Amazon EC2. We use the Hadoop MapReduce framework for performing large-scale computations and HDFS for sharing common data among nodes[11]. We wrote a MapReduce application in order to perform the APK to basic-block conversion portion of the workflow, and we used Hadoop Streaming to interface with our C++ applications, which were responsible for feature hashing and containment calculations. We note that the Hadoop Streaming interface is unable to take advantage of resource management because of the our application is invoked from Hadoop Streaming as an external program, and therefore must distribute data manually to and from HDFS nodes. We also note we are unable to take advantage of other Hadoop optimizations, like compression, intelligent file and resource distribution across nodes, and caching of resources[4]. Many of the tasks required of Juxtapp are easily parellizable tasks which greatly improves performance when dealing with large datasets. As a result, Juxtapp can feature hash, cluster, and analyze containment in a distributed manner which offers great performance increases over the single machine version.

Incremental Update and Increasing Performance The statelessness property of many stages in Juxtapp makes it easy to incrementally process the applications, update their similarity matrix, and analyze them in detail without the need to reprocess all applications under analysis. When creating or updating the pairwise similarity matrix, only values greater than *50%* similarity are stored, making the matrix sparse among dissimilar applications. When a set of m new applications are added to the analysis, the application preprocessing (conversion of APK to XML to basic block) and the feature hashing are inherently incremental, meaning, only the new applications need conversion and feature hashing. As shown in Figure 5.3, with n existing applications and m new applications, updating the existing $n \times n$ similarity matrix A is as follows: 1) compute the $m \times m$ similarity matrix B among the new applications, 2) compute the $n \times m$ similarity matrix C between the set of new applications and the existing ones, and 3) concatenate them together and grow the existing similarity matrix A at appropriate rows and columns to get the new $(n+m) \times (n+m)$ similarity matrix. With the new similarity in B and C, it is also straightforward to update the hierarchical clustering using incremental methods to obtain a new clustering results [65, 44].

5.5 Evaluation

In this chapter, we evaluate the efficacy of Juxtapp. We first introduce our evaluation dataset and describe our experimental setup. Then, we discuss determining experimental parameters and their impact on our results. Finally, we introduce case studies in which we use Juxtapp to detect instances of vulnerable code reuse, known malware, and piracy on Android markets.

5.5.1 Experimental Evaluation Dataset

We evaluate our approach using applications from three different sources. From the official Android Market we obtained 30,000 free Android applications. Additionally, we downloaded 28,159 applications from a third-party Chinese market, Anzhi, a Chinese corporation, which aims to be the central repository for Chinese third-party Android applications. Anzhi was the first third party market to break 10 million users in 2010, and as of 2012 claims to have more than 25 million users worldwide, with over 180 million downloads per month[1]. The 72 malware in our malware dataset came from the Contagio malware dump and other sources [4]. Lastly, we use a set of 95,000 Android applications from the official Android Market to evaluate the performance of Juxtapp [5].

[4] Hadoop Streaming uses program input and output to interface with the Java application.

[5] We obtained a larger dataset of applications in order to show that our technique scales to a large number of applications beyond our evaluation set of applications

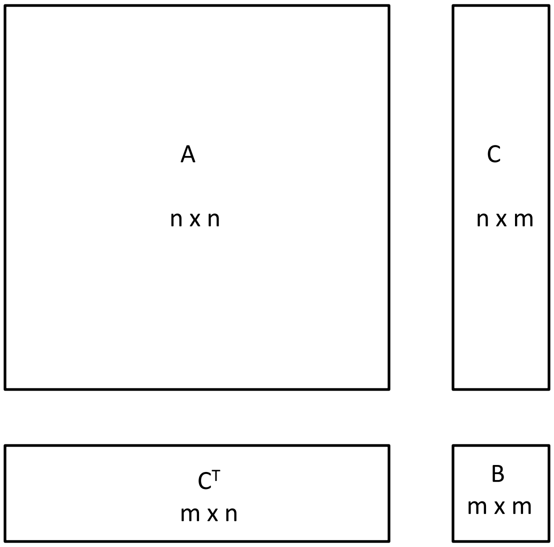


Figure 5.3: Incrementally updating the similarity matrix. A contains similarity values among existing applications, B contains similarity values among the new applications, and C contains similarity values between the new applications and the existing ones.

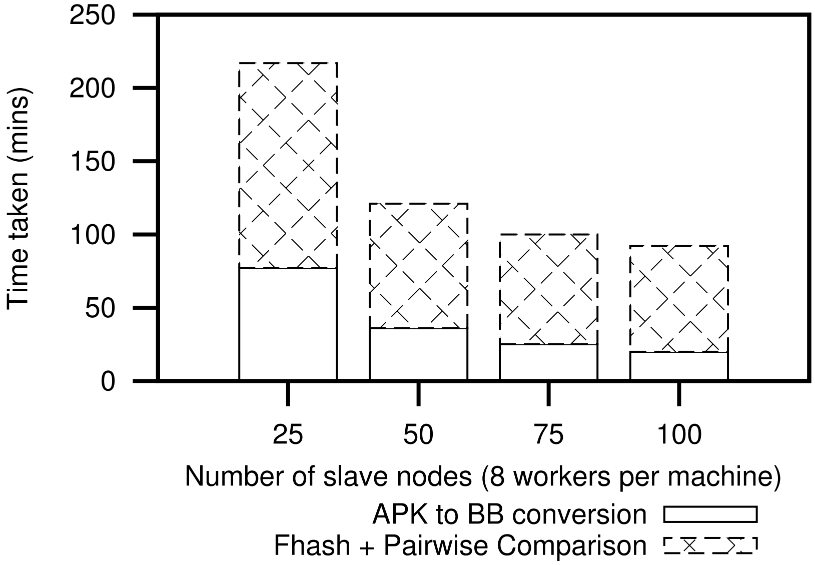


Figure 5.4: The running time of our complete pipeline with various number of workers per cluster on Amazon EC2. Time are measured when processing 95,000 unique Android applications.

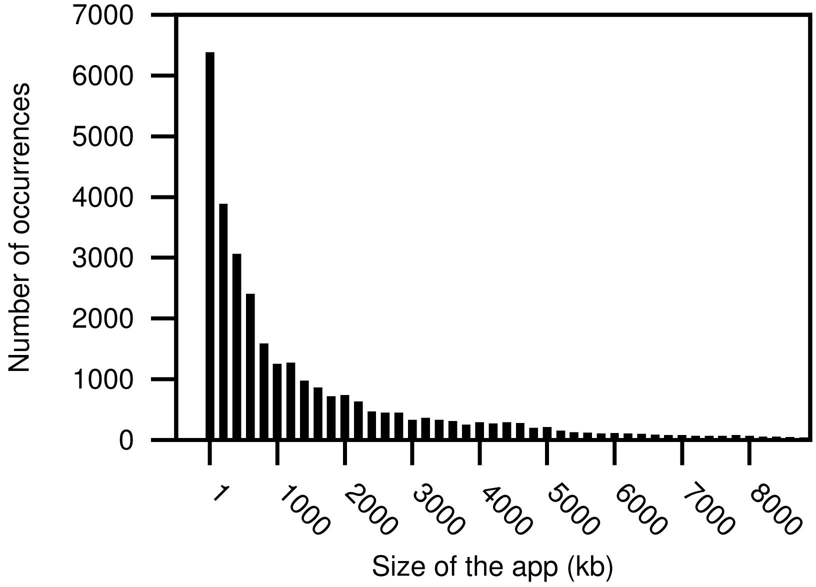


Figure 5.5: Frequency distribution of size of APK files. For better visualization, we do not show the largest 4% of the applications. The largest APK file in our dataset is 52.26MB. The numbers on the x-axis are the lower bounds of the bins, and the size of each bin is 200.

5.5.2 Experimental Setup and Performance

Local experiments, when tractable, such as containment between a small set of applications and our dataset, were run on Ubuntu Linux 2.6.38 with Intel Xeon CPU (8 cores) and 8GB of RAM. When larger experiments were required, such as containment between on-market to off-market applications, and generating pairwise distance matrix, we conducted them on Amazon EC2. For our Amazon EC2 clusters, we used m2.4xlarge instances, which run on Ubuntu Linux 2.6.38-8-virtual with 8 virtual cores and 68.4GB of memory.

We varied the number of nodes running from 25 to 100 and used 8 worker threads per node. Figure 5.4 shows the time required to complete a full run of the entire pipeline, which includes APK to basic block format file conversion, feature hashing, and computation of the pairwise similarity when using 95,000 unique Android applications. At the time of writing, there are around 310,000 Android Applications[16], which demonstrates that Juxtapp scales well.

As we increase the number of nodes, the amount of time required to do analysis becomes gradually dominated by the overhead of parallelization, including the distribution of resources and the recombination of resources in order to combine all of the resulting bit vectors in order to calculate the pairwise distance matrix. In addition, the APK to basic-block and feature hashing stages were parallelizable without any synchronized state, which contributed to significant performance gains as the number of workers increased. Figure 5.4 shows how the overhead of the pairwise comparison approaches a constant overhead as the number of nodes are increased. We speculate that a native implementation in Java using Hadoop would

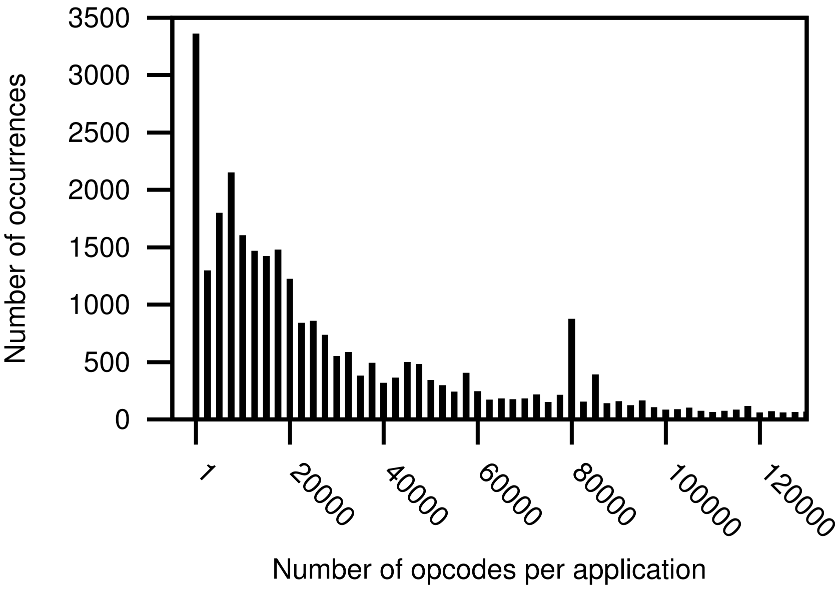


Figure 5.6: Frequency distribution of number of opcodes of applications. For better visualization, we do not show the largest 11.2% of basic block files. The largest file in our dataset has 1,728,196 opcodes. The numbers on the x-axis are the lower bounds of the bins, and the size of each bin is 2500.

reduce the overhead across the board in both the resource distribution and recombination steps.

Incremental Update Performance. Incremental updates of the dataset allow us to continuously process and update our dataset with new market applications without requiring running the entire Juxtapp workflow on our application repository. Table 5.5.2 shows the time required to add from 100 to 7,000 APKs to the dataset. Distribution time is the time required to distribute APKs to worker nodes. This time begins to become dominant as the number of APKs increases. This overhead is caused by not being fully able to take advantage of Hadoop's resource allocation, due to our Hadoop Streaming implementation. Despite this, these numbers show that adding a large number of applications to the comparison repository daily or even multiple times daily is feasible with Juxtapp.

5.5.3 Dataset Statistics

To gain a general understanding of our dataset, we analyzed our collection of 30,000 unique applications as a representative sample of the official Android Market. Figure 5.5 shows the distribution of the sizes of APK files in kilobytes, and Figure 5.6 shows the distribution of the number of opcodes per application. Both distributions are skewed to the right, with APK files having a median size of 724KB and applications having a median number of opcodes of 20,555. The 75th percentile values for APK file sizes and number of opcodes are 2,071kb

# Incr. APKs	Distribution Time	Completion Time
100	0m 36s	5m 11s
500	4m 49s	9m 35s
1000	8m 58s	21m 5s
3000	20m 20s	42m 31s
5000	42m 52s	80m 51s
7000	57m 0s	104m 48s

Table 5.1: The time to incrementally process varying numbers of APKs. Note, distribution time is included to show how file distribution starts to dominate the processing time.

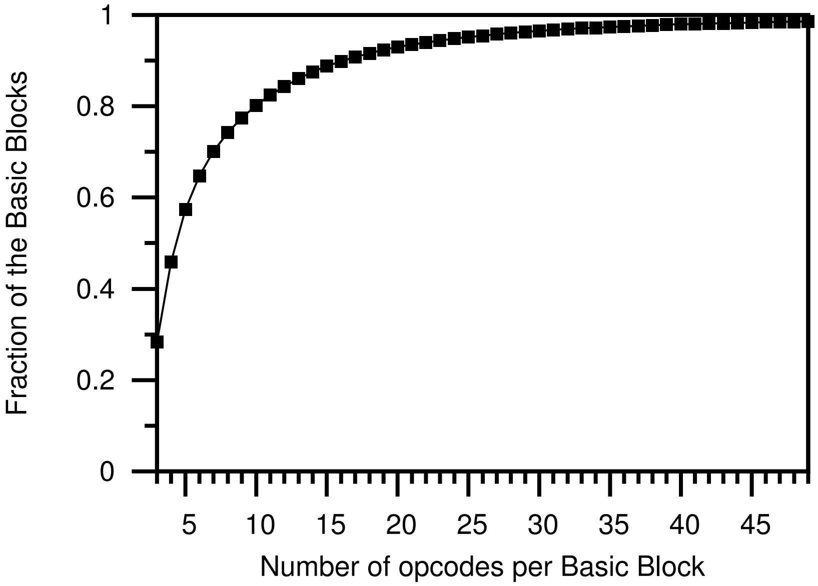


Figure 5.7: Cumulative distribution of the number of opcodes per basic block, using all basic blocks with more than 2 opcodes. The mean is 5.35 opcodes and the median is 2 opcodes, while the largest one contains 35,517 opcodes.

k	Average Jaccard Distance
3	0.939
5	0.969
7	0.980
9	0.984

Table 5.2: Experiment showing the impact of varying k on the Jaccard distance.

and 56,166, respectively. The total file size of these APKs is 50.43GB and total number of opcodes in all applications is approximately 1.45 billion.

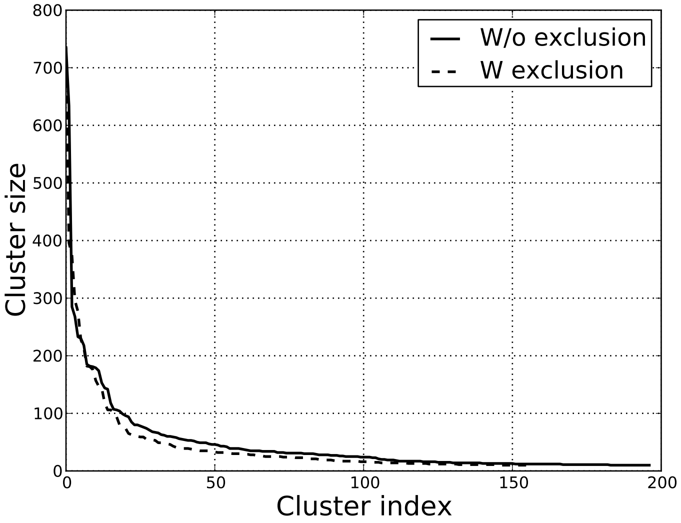


Figure 5.8: The clusters obtained from 30,000 Android application using hierarchical clustering with similarity threshold $t = 0.9$.

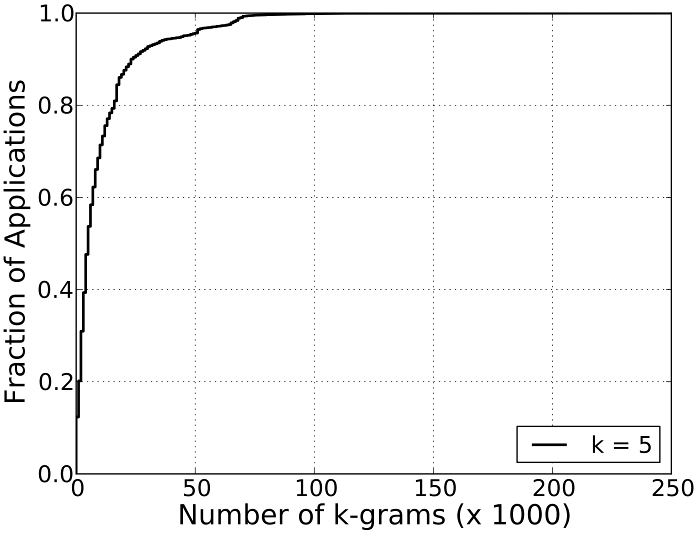


Figure 5.9: Cumulative distribution of the number of unique k-grams extracted from 30,000 Android applications (with $k = 5$).

5.5.4 Determining Experimental Values

Before feature hashing we must choose values for k-gram size k and bitvector size m. We use the 30,000 Android applications to determine their values.

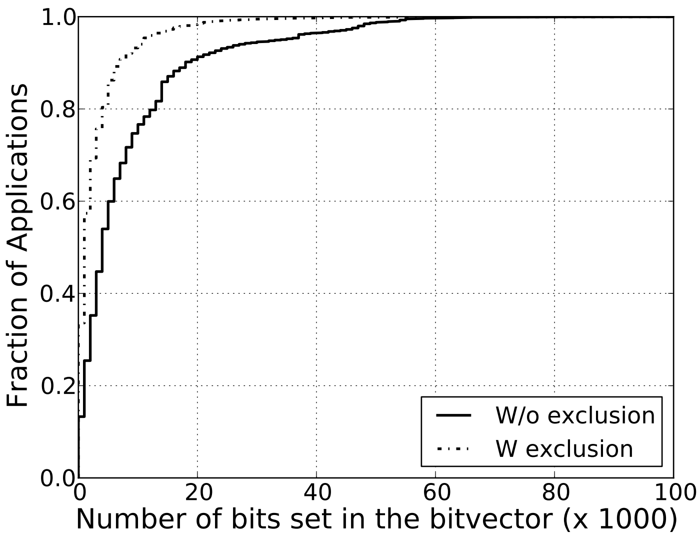


Figure 5.10: Cumulative distribution of the number of bits set in the bitvectors of 30,000 Android applications.

5.5.4.1 Choosing k for our Dataset.

To choose k, we randomly select pairs of applications and evaluate their Jaccard distance to determine how much varying k impacts the average distance between them. Table 5.2 shows varying values of k and the resulting average distance between pairs of randomly sampled 6,000 applications[6]. We repeat the experiment on multiple runs, but see little variance across them. The key intuition is if two applications are chosen at random from our dataset, they are likely to be dissimilar. The table shows that starting from 5, further increasing k has little impact on the distance calculation. Based on this, we chose a value of k to be 5 and performed feature hashing and clustering on our sampled applications. Figure 5.7 shows the cumulative distribution of opcodes per basic block for all basic blocks with more than two opcodes. This indicates that the majority of the basic blocks are dominated by a small value of k, and 5 is an appropriate choice for this dataset.

5.5.4.2 Choosing an Appropriate Bitvector Size.

The bitvector size m strikes the trade-off between efficiency (similarity) computation and approximation error of using bitvectors to represent the sets of k-gram for each application. Ideally, we want size m to be large enough such that few collision occur during feature hashing. Practically, we want size m to be small so that we can efficiently compute similarity between all pairs of hundreds of thousands of applications.

According to [50], we need $m \gg N$, the number of k-grams extracted from an application, so that the Jaccard similarity between two bitvectors is very close to the exact

[6] A distance of 1 indicates no similarity where a distance of 0 indicates identical similarity.

representation of computing the intersection between two k-gram feature sets. In addition, in all of our analysis, we are particularly interested in applications with high similarity, e.g., *application pairs with similarity greater than 50%*. We sparsely store this pairwise matrix and only store values for which the threshold is reached. This optimization yields good similarity results because those excluded applications at the 50% threshold are very unlikely to have similar applications within the comparison dataset.

We use all of the 30,000 applications from the Android Market to determine m. We compute the number of unique 5-gram features that can be extracted from each application, and plot its cumulative distribution from all applications in Figure 5.9. We find N_{90} in the distribution, which represents the threshold in which 90% of all applications' k-gram features are less than this value. We then set $m = 240,007$, a prime that is more than nine times N_{90}, satisfying the condition $m \gg N$ suggested by [50].

We use the following two ways to verify whether $m = 240,007$ is large enough. 1) We do feature hashing with $m = 240,007$ for all 30,000 applications, and count the number of bits set in the bitvector for each application. We plot its distribution in Figure 5.10. We observe that more than 95% of applications have $1/5$ of their bits set in their bitvectors, and more than 90% of applications have only $1/10$ of their bits set. Hence, we do obtain sparse bitvector representation for the majority of applications. 2) We also randomly sample a subset of 1000 applications, compute the pairwise similarity among them using their k-gram feature sets, and compare the similarity values to those computed using their bitvectors. We find that the average difference between them is less than 0.01. With these two observations, we conclude that $m = 240,007$ is suitable for our analysis.

5.5.4.3 Clustering for Application Topology

We use clustering as a way to group similar applications together. We run hierarchical clustering on the 30,000 Android applications using a similarity threshold $t = 0.9$, with and without a core functionality exclusion list applied, respectively. Figure 5.8 shows the cluster size sorted in a descending order. However, as stated before our automatic exclusion list generation did not work as well as we had hoped. Large, common libraries like Admob were automatically pruned, but after investigation we determined that clusters were still dominated by large libraries, not quite capturing the core functionality. We found that clusters with exclusion no longer had application clusters dominated by large libraries, instead they were dominated by smaller libraries. However, the clustering as a whole worked well to detect similarity within a given threshold.

We observe that there are around 200 clusters, each of which has at least 10 applications, and in total there are 9344 applications in those clusters. We find that our clustering identified three unique, commonly occurring patterns. They are:

Same application title, different versions. One cluster contained several versions of the same movie player, which were all responsible for displaying explicit pictures of models. Within the cluster, there were *4* different versions of the same model's movie player, heaven8.

Differing author and functionality, same tool for development. In one example, AppBar is a tool for allowing users to visually create applications for Android without needing to know about the underlying development platform. The platform allows for the addition of sounds, images, twitter feeds, and all sorts of additional widgets. We identified a cluster on the official Android market consisting of *735* applications of this type, ranging from RSS feeds to audio programs.

Multiple apps from an author, different underlying functionality. A common pattern is for a developer to make a framework for creating applications and then reusing the applications in a variety of contexts. For instance, the company BrightAI produces a variety of applications related to sports. One such cluster contained *28* different applications, all by BrightAI, but with different application purposes.

We stress that clustering with a high threshold (say 90% match) can miss similarities among applications with a large amount of code in common. For example, multiple versions of the same application across versions may have additional libraries added or removed, thus reducing the amount of similarity across this threshold. Simply put, this works well for classifying very similar applications on a global view, while containment is useful for comparing individual samples to see what fraction of code they have in common. Lowering the threshold can capture similar applications of different sizes, at the cost of additional noise in each cluster. Meaning, applications may be clustered based on large libraries within each application, as opposed to their core functionality.

5.5.5 Case Studies

Previous work on studying Android applications[37] has shown that developers copy and paste code snippets from popular programming web sites into their own code, without understanding the potential security risks posed by blindly copying code.

Recently, Google announced an In-Application Billing API along with a sample application which demonstrates how the purchasing protocol works[14]. Several security warnings accompany the document, including statements regarding how developers should obfuscate their code, protect their purchasable content, and verify purchases on a remote server. We show how Juxtapp can not only detect applications in the Android Market that copied this sample code, but we also show how we can detect other known source code-related vulnerabilities in the market using our architecture.

5.5.5.1 Reuse of Vulnerable Code

In this chapter, we examine two cases of vulnerable code reuse of sample code provided by Google: In-Application Billing and the License Verification Library. We show that Juxtapp can quickly and efficiently reduce the set of potentially vulnerable applications and detect vulnerable code reuse in Android Applications.

In-Application Billing. Google In-Application Billing (IAB) is a library provided for developers to include so that their customers can sell digital content within their application, while letting Google handle authentication and credit card purchases[9]. For security reasons, Google advises that developers use obfuscation in order to make the code more difficult to understand for an adversary and they also recommend that developers perform verification on a remote server.

However, the sample code provided by Google is not obfuscated and performs verification of a purchase on the device. The left side of Figure 5.11, Line 231, shows the potential single point of attack. Meaning, if a developer can rewrite the statement to negate the condition, or force it to be true in some other way, the application will skip verification and allow the current user access.

In order to detect a potential attack, we analyzed the containment between the IAB sample code and the 30,000 applications in our dataset. We set a threshold that at least 70% of the IAB sample code must be in the application before further exploration.

Running containment between the sample IAB code and the Android Market applications took *1.5 minutes*, and we detected **295** applications containing 70% of the IAB code. Other researchers used these applications to demonstrate that they could use the tool they developed for application rewriting to automatically exploit a vulnerability to get virtual goods for free [21]. Of those that used a significant portion of the sample code, **174** were vulnerable, while 65 use off-device/JNI verification and 56 were inoperable after rewriting. Our results show that Juxtapp is a fast way to quickly analyze large sets of applications for vulnerabilities caused by code reuse.

License Verification Library. The License Verification Library (LVL) is a library provided by Google in order to allow developers to query the Android Market at runtime in order to determine if a user is licensed to use a particular application[10]. Similar to IAB, Google provides sample code which encourages developers to obscure their code and ensure that single points of attack are protected. The sample code uses caching in order to prevent having to contact the Android Market every time the user invokes the application. However, the right side of Figure 5.11, Line 133, shows the potential vulnerability. This line could be rewritten to negate the condition, or to check another condition, making this a single point of failure, allowing a clever attacker to use the library without a license.

We executed containment on 30,000 using the Google LVL sample code to guide the search. For this experiment, we detected 272 potential candidates, **182** of which had 90% of the code, and **90** more, with at least 70% of the sample code. It took about *2 minutes* to analyze the dataset. Of the potentially vulnerable candidates, **239** of the 272 applications had the vulnerable pattern in their code. We manually verified the results in order to be assured that the pattern was in the code. Our analysis took about 10 minutes with script assistance responsible for opening each document which allowing the analyst to determine if the pattern exists, without the task of manually opening each file. Of those detected, some had obfuscated class and method names, but Juxtapp was still able to detect similarity. A

```
boolean verify(...) {
if (!sig.verify(
       Base64.decode(signature))){
    return false;
}
return true;
```

```
void checkAccess(...) {
// skip asking market if cached license
if (mPolicy.allowAccess()) {
    callback.allow();
} else {
    //verification code
```

Figure 5.11: The code on the top shows the vulnerable code present in the In-Application Billing Example Code `Security.java`. On the bottom is the point of vulnerability within the License Verification Library sample code `LicenseChecker.cpp`.

few of the applications which were not vulnerable omitted the check for a cached response so that each time the user wished to use the application, a license must be granted[7].

Malware	Instances Found	Distinct New Carriers Found	Malware BB Size
GoldDream	25	13	1,898
DroidKungFu	6	0	5,357
DroidKungFu2	2	0	375
zsone	1	0	280
DroidDream	0	0	2,526
Total	**34**	**13**	-

Table 5.3: Number of instances of each kind of malware found in the Anzhi Market dataset. Also shown are the distinct new carriers discovered in our dataset.

5.5.5.2 Android Malware

The Android Market place has recently experienced an influx of malware. Google has responded by exercising its remote application removal ability, that is, if Google determines an application is malicious or untrustworthy, it can remotely push a command to remove the application from affected devices[7]. In fact, as of August 2011, users are 2.5 times more likely to encounter malware on their mobile devices than only 6 months ago, and it is estimated that as many as 1 million users have been exposed to mobile malware[15][19]. We suspected that unregulated, 3rd party markets will have a higher incidence of malware and indeed our results and research published concurrently with this chapter shows that third party markets do have a high incidence of malware[88, 89].

Containment between Anzhi Market and Malware. In order to evaluate whether third party markets contain known malware, we select a subset of 5 malware applications from our dataset, which represents some of the most prolific, well-known malware. They include: DroidDream, DroidKungFu1/2, zsone and GoldDream. Each malware sample had a manual exclusion list applied, that is, using widely available malware analysis, we excluded common

[7] Note: While this code is technically safe, Google advises against this because applications will be unusable when a user does not have Internet access.

code from malware such as advertising libraries and common utilities which contribute nothing to the uniqueness of the code. The exclusion list can also be generated in a semi-automatic manner. As we stated our core functionality did not generate ideal exclusion lists. In practice, we did automatic generation, and then pruned or added to the list with analyst-driven expertise. Juxtapp can simplify this task for the analyst by first attempting to find a matching carrier application which is the same as the malware and then removing all similar code between the matching applications. The resulting code would be an exclusion list with only the differences between the applications remaining, which in this case is the malware exclusion list.

Table 5.3 shows that we were able to detect 34 malware applications in the off-market dataset. We define a *carrier application* to be a legitimate application that has been modified to contain a malicious payload. The experiment took around 10 minutes to complete. Among those that matched we noticed a very high incidence of code reuse ranging from 93%-100%. The lower percentage matching shows that the technique is amenable to code mutations and variants. When investigating those with lower percentages, we noted that variants often changed file paths, reworked small amounts of code, changed exploit names, etc., and a 100% match indicated, with high probability, that the two pieces of malware are identical and indeed, when investigated the samples matched.

When evaluating the samples, we also consider the ratio of the size of the malware sample compared to the size of the container application in order to remove false positives. As stated before, this ratio is the size of the number of features of the two applications under comparison. An application that has a high percentage match, but a low ratio indicates that it is likely a false positive due to the high density of the bit vector of the large application in comparison to the smaller application. A low ratio (e.g., .8x, 1x, 2x) indicates similar orders of magnitude among the code sample, where a higher ratio (e.g., 40x, 50x, etc.) indicates that the reported matching is likely a false positive due to the density of the bitvector representing the larger application. For instance, when evaluating our dataset for containment of GoldDream, we evaluated both the ratio and percentage match in order to eliminate false positives. We found that the highest percentage match outside of the range discussed previously (93%-100%) was 73% with a code ratio of 20 times greater than that of the sample, indicating a false positive. This sort of clear demarcation allows us to quickly and easily identify malware samples and discard false positives. Some malware found in the Anzhi market matched our sample malware dataset with little variation in code between them. However, other matched malware was significantly different from our evaluation set and we show how we can detect new variants, with new malware carriers using Juxtapp. Most of the minor changes were related to class and package names. However, Table 5.3 shows that we found 13 unique carriers of the GoldDream malware in our dataset. Meaning, of these we found **13** previously unknown to us, distinct applications in our evaluation dataset, which were all different types of games that had been repackaged with the GoldDream malware. Of the 13 significant variants, one of them was found 12 times in the Market with only very small differences between them, so we consider this to be one instance. The differences between very closely matching variants were constant strings, relating to the storage of various dropped files that the malware uses, along with variation in the class and package name, but little else. Our results confirm our suspicion that third party markets house known malware and that Juxtapp can be used to find known malware and previously unknown, new variants of them.

Identifying Contaminated Code. As a further step in analysis and verification of malware, Juxtapp can identify the commonality between applications and exclude those features that are contained within both applications and output the resulting code. By excluding the code that caused the match, an analyst can quickly discover which portions of the code were modified or injected into the application. For instance, when conducting the GoldDream experiments above, we could identify the carrier applications because the GoldDream portions of the application are removed, due to sharing similar code across variants. This allowed us to quickly and easily identify the main, non-infected, components of the applications.

Containment between Android Market and Malware. We evaluated containment between 63 malware samples to the 30,000 collected from the official Android Market. The experiment took 19 minutes to execute locally.

Juxtapp did not detect any instances of known malware on the Android Market. This result is unsurprising given that Google has been vigilant about removing malware once it is found, banning the associated account, and issuing remote removal[20].

However, as expected, Juxtapp was able to detect the original application that the malware sample had been repackaged with in order to trick users into downloading. That is, a subset of our samples were repackaged with legitimate applications. Table 5.4 shows the Android applications we were able to detect using the malware sample.

Application File Name	Features	Name	Repackaged with
com.codingcaveman.solotrial.apk	4,272/4,831	Guitar Solo Lite	DroidDream.1
it.medieval.blueftp.apk	19,597/18,946	Bluetooth File Transfer	DroidDream.2
com.tencent.qq.apk	28,712	Tencent QQ Messaging	PJApps
de.schaeuffelhut.android.openvpn.apk	2,009	OpenVPN Settings	DroidKungFu

Table 5.4: Juxtapp is able to detect the original (and versions) of the application which was repackaged when compared to our malware dataset. Multiple features indicate multiple versions in our dataset.

5.5.5.3 Piracy and Application Repackaging

In addition to vulnerable code and malware on the Android markets, piracy, especially among games, has become a major problem for developers. Android applications are often pirated by rogue authors, which remove copy protection and replace developer revenue mechanisms such as advertising libraries. In order to examine the third party market Anzhi for piracy, we downloaded and paid for the two applications mentioned in a Guardian article about Android privacy[6]: 1) Chillingo's The Wars; 2) Neolithic Software's Sinister Planet. We compared these applications against the 28,159 applications in the Anzhi market, which took around 19 minutes to execute locally.

We found no instances of the Sinister Planet program being pirated on a third-party market. However, we found **3** pirated versions of Chillingo's The Wars, being marketed by the company Joy World, the same company accused of piracy in the article. Each of the pirated versions has **71%** code in common with the original application. There were no false positives in either case.

Despite the fact that the legitimate Wars program is unobfuscated, the Joy World version is obfuscated with methods and classes renamed. Additionally, we found that the pirate had added advertising libraries to the application which were not present in the original version. So, even in light of significant obfuscation and additional code added, we were still able to detect similarity showing that Juxtapp handles perturbations in code well. Finally, the pirate did not remove the company name of the original producer of The Wars. Notably, the original maker, Chillingo, releases names under Deluxeware as well and this name *remains* in all pirated versions of the code. We found that advertising and other libraries were added to the pirated versions to generate revenue for the pirate. In this chapter, we have shown that Juxtapp works well in detecting code reuse in Android applications. Namely, we've shown that instances of piracy, buggy code reuse, and known malware, even with obfuscation, can be detected by our tool.

5.6 Related Work

DroidMOSS uses a technique similar to ours has been independently developed by Zhou *et al.* [88]. While they focused on detecting repackaged applications, we applied our technique and show that it is effective to detect repackaged applications, buggy code reuse and known malware. In addition, we implemented the technique on a distributed infrastructure using Amazon MapReduce, which enables us to analyze a much larger application corpus.

DroidRanger, another system by Zhou *et al.* is a system designed to detect known malware and unknown malware in both the official Android market and third party markets[89]. The primary difference in our tools is that Juxtapp primarily uses static features and bytecode semantic information to detect known malware and variants, where DroidRanger uses heuristic and known malware-based filtering to discover new malware and detect new malware based on known exploit vectors. For known malware detection, their system uses behavioral-based detection in which they filter based on dangerous permissions malware has been known to use, like `SEND_SMS`. Additionally, they include information from the manifest (like listening for SMS messages), semantic information from the bytecode (strings and numbers), and structural layout of the applications packages and classes. Secondly, they design a heuristic model for detecting malware which monitors dynamic loading of code and also has dynamic execution monitoring including Linux syscalls associated with known exploits. Unlike Juxtapp, the performance and scalability of DroidRanger is not discussed.

For large-scale malware analysis, Jang *et al.* [50] developed *BitShred*, a system for large-scale malware triage and similarity detection based on feature hashing. However, they focus on the technique as a contribution and classify x86 malware, whereas we apply similar techniques, with domain specific knowledge in order to find a variety of code reuse in

Android marketplaces. Instead of using boolean features, Gao *et al.* [41] and Hu *et al.* [49] use features based upon isomorphisms between control flow and function call graphs of the program. While these work primarily focus on techniques to compare and index malware, our work is focused on techniques to determine similarity among Android applications and conduct deep security analysis.

Winnowing, a fuzzing hashing technique that selects a subset of features from a program for analysis, has been widely used for code similarity analysis[24] and plagiarism detection[70]. However, the winnowing algorithm requires calculating set inclusion, which is expensive when comparing many features.

A variety of approaches for static code clone detection have been proposed in the programming language literature for refactoring, finding bugs, and better understanding of the code [40, 55, 59, 51]. All those techniques can be applied into our framework to further improve the accuracy and robustness our approach.

5.7 Conclusion

In this chapter, we presented Juxtapp, a scalable architecture for detecting code reuse in Android applications. Our architecture is implemented in Hadoop and we ran it on Amazon EC2. We evaluated the efficacy of Juxtapp in detecting vulnerable code reuse, known malware, and piracy in a dataset of 58,000 applications from Android marketplaces. In the evaluation of vulnerable code reuse, we found that many developers did not modify the sample code significantly from the Google-provided libraries, which left applications vulnerable to a variety of rewriting attacks. Furthermore, we used Juxtapp to discover **34** instances of malware, including *13* variants of the GoldDream malware which we found to be using a variety of games as carrier applications. Finally, we found *3* instances of piracy in which a game was victim to removal of copy protection and the addition code to include a multitude of advertising libraries which benefit the pirate. Our findings show that Juxtapp is a valuable architecture in detecting application similarity and code reuse in Android applications.

5.8 Acknowledgments

Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications originally appeared at **DIMVA 2012**. It is an original work by Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen and Dawn Song. We would like to thank Adrienne Felt and Chris Grier for their insights with this chapter.

Chapter 6

Conclusion

New architectures require careful examination of security properties in order to assess and contain new threats. In light of this, emerging technologies, such as web APIs, medical devices, and applications on mobile phones, are a new security landscape that has recurring security problems. In this dissertation, we develop new techniques to analyze several emerging applications for security vulnerabilities. We utilized techniques including: dynamic symbolic execution, binary analysis and reverse engineering, and wide scale application comparison and classification.

First, we created *Kudzu*, a system for symbolic execution of JavaScript. We detailed how our tool systematically explores client-side JavaScript programs. We explored both the value space and event space of these applications in order to discover client-side validation vulnerabilities.

We used *Kudzu* to further examine how new HTML5 primitives. These new primitives are those used to create complex client-side JavaScript applications. We show how they may be abused by revealing attacks against *postMessage* and the client-side persistent storage API.

Then, we explored the area of medical device security. We reverse engineered the CardiacScience G3 Plus Automated External Defibrillator, a publicly available, wide-spread medical device. We demonstrated 4 vulnerabilities in the device and its software. We then offered suggestions to guide future medical device design based on the results of our investigation. We demonstrated that this new frontier of life-critical devices, especially as they become more advanced, must be secure and reviewed for security threats.

Finally, we developed *Juxtapp*, a system for discovering similarity among Android applications. Juxtapp is a scalable, efficient and capable of quickly and accurately detecting similarities among Android applications. We used *Juxtapp* to detect piracy, code reuse, and known malware in third party markets.

We demonstrate that these techniques are useful at discovering and/or preventing attacks, in their respective application domains.

Bibliography

[1] Anzhi android market. http://www.anzhi.com/aboutus.php.

[2] Blaze: Binary analysis for computer security.

[3] Cardiopulmonary Resuscitation (CPR) Statistics. American Heart Association. http://www.americanheart.org/presenter.jhtml?identifier=4483.

[4] Contagio malware dump. http://contagiodump.blogspot.com/.

[5] Dalvik virtual machine. http://www.dalvikvm.com/.

[6] Developers express concern over pirated games on android market. http://www.guardian.co.uk/technology/blog/2011/mar/17/android-market-pirated-games-concerns/.

[7] Exercising our remote application removal feature. http://android-developers.blogspot.com/2010/06/exercising-our-remote-application.html.

[8] External defibrillator improvement initiative. FDA, Center for Devices and Radiological Health. http://www.fda.gov/downloads/MedicalDevices/ProductsandMedicalProcedures/CardiovascularDevices/ExternalDefibrillators/UCM233824.pdf.

[9] Google in-app billing. http://developer.android.com/guide/market/billing/index.html.

[10] Google license verification library. http://developer.android.com/guide/publishing/licensing.html.

[11] Hadoop. http://hadoop.apache.org/.

[12] Hash functions. http://www.cse.yorku.ca/ oz/hash.html.

[13] IDA Pro. Hex-Rays SA, http://www.hex-rays.com/idapro/.

[14] In-app billing. http://developer.android.com/guide/market/billing/index.html.

[15] Mobile threat report. https://www.mylookout.com/mobile-threat-report/.

[16] Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps/.

[17] Proguard. http://developer.android.com/guide/developing/tools/proguard.html.

[18] Safeseh. Microsoft, http://msdn.microsoft.com/en-us/library/9a89h429(VS.80).aspx.

[19] Up to a million android users affected by malware, says report. http://www.linuxfordevices.com/c/a/News/Lookout-malware-report-2011/.

[20] Update: Security alert: Droiddream malware found in official android market. http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/.

[21] Freemarket: Shopping for free in android applications. In *Extended Abstract, to appear NDSS*, 2012.

[22] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580, New York, NY, USA, 2011. ACM.

[23] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *International Symposium on Software Testing and Analysis*, 2008.

[24] Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *Proceedings of the USENIX Annual Technical Conference*, 1998.

[25] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[26] Adam Barth, Collin Jackson, and William Li. Attacks on JavaScript mashup communication. In *Web 2.0 Security and Privacy*, 2009.

[27] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, 2008.

[28] Hristo Bojinov, Elie Bursztein, and Dan Boneh. XCS: Cross channel scripting and its impact on web applications. In *CCS*, 2009.

[29] J. Richard Büchi and Steven Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. *Mathematical Logic Quarterly*, 34(4):337–342, 1988.

[30] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communication Security*, Alexandria, VA, October 2007.

[31] Paul S Chan, Harlan M Krumholz, John A Spertus, Philip G Jones, Peter Cram, Robert A Berg, Mary Ann Peberdy, Vinay Nadkarni, Mary E Mancini, and Brahmajee K Nallamothu. Automated External Defibrillators and Survival After In-Hospital Cardiac Arrest. *JAMA : The Journal of the American Medical Association*, 304(19):2129–2136, November 2010.

[32] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of MobiSys*, 2011.

[33] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *PLDI*, 2009.

[34] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley and Sons, 2000.

[35] FDA. FDA Warns Users about Faulty Components in 14 External Defibrillator Models, April 2010. http://www.fda.gov/NewsEvents/Newsroom/Press Announcements/ucm209874.htm.

[36] CDRH preliminary internal evaluations — volume I: Preliminary Report and Recommendations, August 2010. http://www.fda.gov/downloads/AboutFDA/ CentersOffices/CDRH/CDRHReports/UCM220784.pdf.

[37] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of ACM CCS*, 2011.

[38] Ian Fette. Hello HTML5. `http://gearsblog.blogspot.com/2010/02/hello-html5.html`.

[39] Kevin Fu. Trustworthy medical device software. Washington, DC, 2011. IOM (Institute of Medicine), National Academies Press.

[40] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[41] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[42] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security*, February 2008.

[43] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Usenix Security*, 2009.

[44] Ibai Gurrutxaga, Olatz Arbelaitz, Jos I. Martn, Javier Muguerza, Jes M. Prez, and Iigo Perona. Sihc: A stable incremental hierarchical clustering algorithm. In *In Proceedings of ICEIS*, 2009.

[45] Daniel Halperin, Thomas S. Heydt-Benjamin, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Security and Privacy for Implantable Medical Devices. *IEEE Pervasive Computing*, 7(1):30–39, 2008.

[46] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the 29th Annual IEEE Symposium on Security and Privacy*, May 2008. Outstanding Paper Award.

[47] Robert Hansen. XSS cheat sheet. `http://ha.ckers.org/xss.html`.

[48] Billy Hoffman and Bryan Sullivan. *Ajax Security*.

[49] X. Hu, T. cker Chiueh, and K. G. Shin. Large-scale malware indexing using function call graphs. In *Proceedings ACM CCS*, 2009.

[50] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of ACM CCS*, 2011.

[51] Lingxiao Jiang, G. Misherghi, Zhendong Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE*, 2007.

[52] Ulrich Kahn, Klaus Kursawe, Stefan Lucks, Ahmad-Reza Sadeghi, and Christian Stble. Secure Data Management in Trusted Computing. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 324–338. Springer Berlin / Heidelberg, 2005.

[53] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *30th International Conference on Software Engineering (ICSE)*, May 2009.

[54] KilianWeinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of ICML*, June 2009.

[55] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. Mecc: memory comparison-based clone detector. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 301–310, New York, NY, USA, 2011. ACM.

[56] Amit Klein. DOM based cross site scripting or XSS of the third kind. Technical report, Web Application Security Consortium, 2005.

[57] Amit Klein. Temporary user tracking in major browsers and cross-domain information leakage and attacks, 2008. `http://www.trusteer.com/sites/default/files/Temporary_User_Tracking_in_Major_Browsers.pdf`.

[58] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7, December 2006.

[59] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3), 2006.

[60] Michael Martin and Monica S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *17th USENIX Security Symposium*, 2008.

[61] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In *Proceedings of the International Conference on Web Engineering*, 2008.

[62] Content Security Policy. `https://wiki.mozilla.org/Security/CSP/Spec`.

[63] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *Proceedings of 29th IEEE Symposium on Security and Privacy*, 2008.

[64] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.

[65] Nachiketa Sahoo, Jamie Callan, Ramayya Krishnan, George Duncan, and Rema Padman. Incremental hierarchical clustering of text documents. In *In Proceedings of CIKM*, 2006.

[66] Samy. I'm popular. Description of the MySpace worm by the author, including a technical explanation., Oct 2005.

[67] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Stephen McCamant, Feng Mao, and Dawn Song. A symbolic execution framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[68] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[69] Sam Schillace. Default https access for gmail. `http://gmailblog.blogspot.com/2010/01/default-https-access-for-gmail.html`.

[70] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD/PODS Conference*.

[71] Jignesh S Shah and William H Maisel. Recalls and Safety Alerts Affecting Automated External Defibrillators. *JAMA: The Journal of the American Medical Association*, 296(6):655–660, August 2006.

[72] Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alex Smola, Alex Strehl, and Vishy Vishwanathan. Hash kernels. In *Proceedings of AISTATS'09*, 2009.

[73] Dawn Song, David Brumley, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *In Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[74] Dawn Xiaodong Song, Sergey Berezin, and Adrian Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1/2):47–74, 2001.

[75] Same origin policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[76] Michael Sutton. The Dangers of Persistent Web Browser Storage. `www.blackhat.com/blackhat-dc-09-Sutton-persistent-storage.pdf`, 2009.

[77] TwitPwn. DOM based XSS in Twitterfall. 2009.

[78] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[79] W3C. HTML 5 specification. `http://www.w3.org/TR/html5/`.

[80] W3C. Web SQL Database. `http://dev.w3.org/html5/webdatabase/`.

[81] W3C. Web Storage. `http://dev.w3.org/html5/webstorage/`.

[82] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. In *Proceedings of Duplication, Redundancy, and Similarity in Software*, 2007.

[83] Dolores R. Wallace and D. Richard Kuhn. Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data. *International Journal of Reliability Quality and Safety Engineering*, 8:351–372, 2001.

[84] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the International symposium on Software testing and analysis*, 2008.

[85] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *Proc. of 16th European Symposium on Research in Computer Security (ESORICS)*, 2011.

[86] XML path language 2.0. `http://www.w3.org/TR/xpath20/`.

[87] Jay Yarow and Jon Terbush. Android is totally blowing away the competition. http://www.businessinsider.com/chart-of-the-day-android-is-taking-over-the-smartphone-market-2011-11.

[88] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Droidmoss: Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, 2012.

[89] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, February 2012.