UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik

Precise Detection of Injection Attacks in Real-world Applications

Doctoral Thesis

Submitted to the faculty of computer science and mathematics in fulfillment of the requirements for the degree of Doctor of Engineering (Dr.-Ing.) in Software Engineering

von/by

Gebrehiwet Biyane Welearegai

angefertigt unter der Leitung von/supervised by

Prof. Dr.-Ing. Christian Hammer

begutachtet von/reviewers

Prof. Dr.-Ing. Christian Hammer

Prof. Dr.-Ing. Ben Hermann

Date of oral exam: 2023-11-10

Abstract

Code injection attacks like the one used in the high-profile 2017 Equifax breach, have become increasingly common, ranking at the top of OWASP's list of critical web application vulnerabilities. The injection attacks can also target embedded applications running on processors like ARM and Xtensa by exploiting memory bugs and maliciously altering the program's behavior or even taking full control over a system. Especially, ARM's support of low power consumption without sacrificing performance is leading the industry to shift towards ARM processors, which advances the attention of injection attacks as well.

In this thesis, we are considering web applications and embedded applications (running on ARM and Xtensa processors) as the target of injection attacks. To detect injection attacks in web applications, taint analysis is mostly proposed but the precision, scalability, and runtime overhead of the detection depend on the analysis types (e.g., static vs dynamic, sound vs unsound). Moreover, in the existing dynamic taint tracking approach for Java-based applications, even the most performant can impose a slowdown of at least 10–20% and often far more. On the other hand, considering the embedded applications, while some initial research has tried to detect injection attacks (i.e., ROP and JOP) on ARM, they suffer from high performance or storage overhead. Besides, the Xtensa has been neglected though used in most firmware-based embedded WiFi home automation devices.

This thesis aims to provide novel approaches to precisely detect injection attacks on both the web and embedded applications. To that end, we evaluate JavaScript static analysis frameworks to evaluate the security of a hybrid app (JS & native) from an industrial partner, provide *RIVULET* – a tool that precisely detects injection attacks in Java-based real-world applications, and investigate injection attacks detection on ARM and Xtensa platforms using hardware performance counters (HPCs) and machine learning (ML) techniques.

To evaluate the security of the hybrid application, we initially compare the precision, scalability, and code coverage of two widely-used static analysis frameworks—WALA and SAFE. The result of our comparison shows that SAFE provides higher precision and better code coverage at the cost of somewhat lower scalability. Based on these results, we analyze the data flows of the hybrid app via taint analysis by extending the SAFE's taint analysis and detected a potential for injection attacks of the hybrid application.

Similarly, to detect injection attacks in Java-based applications, we provide *Rivulet* which monitors the execution of developer-written functional tests using dynamic taint tracking. *Rivulet* uses a white-box test generation technique to re-purpose those functional tests to check if any vulnerable flow could be exploited. We compared *Rivulet* to the state-of-the-art static vulnerability detector Julia on benchmarks and *Rivulet* outperformed Julia in both false positives and false negatives. We also used Rivulet to detect new vulnerabilities.

Moreover, for applications running on ARM and Xtensa platforms, we investigate *ROP*[1] attack detection by combining HPCs and ML techniques. We collect data exploiting real-world vulnerable applications and small benchmarks to train the ML. For ROP attack detection on ARM, we also implement an online monitor which labels a program's execution as benign or under attack and stops its execution once the latter is detected. Evaluating our ROP attack detection approach on ARM provides a detection accuracy of 92% for the offline training and 75% for the online monitoring. Similarly, our ROP attack detection on the firmware-only Xtensa processor provides an overall average detection accuracy of 79%.

Last but not least, this thesis shows how relevant taint analysis is to precisely detect injection attacks on web applications and the power of HPC combined with machine learning in the control flow injection attacks detection on ARM and Xtensa platforms.

[1] ROP represents both ROP and JOP

Acknowledgment

At this point, I would like to thank God and all those who have helped me in the completion of this thesis. This work would not have been possible without their help and support.

First, I would like to thank my supervisor, *Prof. Dr.-Ing. Christian Hammer*, for his endless support and guidance. The quality of feedback and ideas that he provides have reshaped my thinking process toward solving a problem and made me a better researcher. Apart from academic support, he also helped me with several bureaucratic procedures. Additionally, I admire him for providing such an energetic and stress-free research environment. Besides, I would like to thank *Dr.-Ing. Abhishek Tiwari* for his constructive feedback on my thesis draft. Similarly, thanks to all of my co-authors and colleagues for providing a positive working environment.

Next, I would like to express my gratitude to my Wife *Birhan Aregawi*, for her unconditional support, encouragement, and love; without her, I would not have come this far. She helped me to focus on my thesis by handling most of the tasks related to children and other family stuff. I was continually amazed by her patience through the ups and downs of my research. Besides, I can not forget my angels, *Amanda* and *Yared*, who add more love and fondness to the family all the time.

Last but not least, I would like to thank all of my family and friends who were continuously motivating me to successfully finish my thesis. They have always helped me to overcome my descents and provided me their valuable advice for important decisions. Especially, I am extremely grateful to my father *Biyane Welearegai* and my mother *Werknesh Tsama* for their unconditional love and support. Without them, I would certainly not stand where I stand today. I couldn't forget the ups and downs they face in their life for me to reach this stage.

Contents

x

List of Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
ARM	Advanced RISC Machine
AST	Abstract Syntax Tree
BL	Branch-with-Link
BLX	Branch-with-Link and Exchange
BR	Branch Regulation
CFA	Control Flow Analysis
CFG	Control Flow Graph
CPU	Central Processing Unit
CPSR	Current Program Status Register
CVE	Common Vulnerabilities and Exposures
DBI	Dynamic Binary Instrumentation
DOM	Document Object Model
ES	ECMAScript
FPGA	Field-Programmable Gate Array
HPC	Hardware Performance Counter
HTML	HyperText Markup Language
IFDS	Inter-procedural, Finite, Distributive Subset
MVC	Model View Controller
IoT	Internet of Things
ISA	Instruction Set Architecture
JOP	Jump Oriented Programming
JSAI	JavaScript Abstract Interpreter
JSON	JavaScript Object Notation
LR	Link Register
LSA	Loop Sensitivity Analysis
ML	Machine Learning
OGNL	Object-Graph Navigation Language
OWASP	Open Web Application Security Project
PC	Program Counter
PMI	Performance Monitoring Interrupt
PMU	Performance Monitoring Unit
RBF	Radial Basis Function
RCE	Remote Code Execution
RISC	Reduced Instruction Set Computing
RIVULET	RIVULET Reveals Injection VUlnerabilities by Leveraging Existing Tests
ROP	Return Oriented Programming
RQ	Research Question

SAFE	Scalable Analysis Framework for ECMAScript
SCS	Shadow Call Stack
SLOC	Subject Line of Code
SP	Stack Pointer
SQL	Structured Query Language
SQLI	SQL Injection
SVM	Support Vector Machine
TAJS	Type Analyzer for JavaScript
TLB	Translation Look-aside Buffer
URL	Uniform Resource Locator
WALA	T. J. WAtson Libraries for Analysis
XSS	Cross-Site Scripting

List of Figures

List of Tables

Chapter 1

Introduction

In the high-profile 2017 *Equifax* attack, millions of individuals' private data was stolen, costing the firm nearly one and a half *billion* dollars in remediation efforts [130]. This attack leveraged a *code injection* exploit in Apache Struts (CVE-2017-5638) and is just one of over 9,711 similar code injection exploits discovered in recent years in popular software [100]. Code injection vulnerabilities have been exploited in repeated attacks on US election systems [143, 51, 95, 24], in the theft of sensitive financial data [132], and in the theft of millions of credit card numbers [78]. In the past several years, code injection attacks have persistently ranked at the top (#1 until 2021) of the Open Web Application Security Project's (OWASP) top ten most dangerous web flaws [105]. SQL injection and cross-site scripting (XSS) attacks are the most widely used injection attacks in web applications.

The injection attack is also targeting the fast-growing mobile and internet of things (IoT) applications by exploiting memory bugs and maliciously altering the program's behavior or even taking full control over a system. Memory exploitation can be done by writing new machine code into the vulnerable program's memory or by reusing existing code. The latter is imperative when a protection technique known as W $\oplus$ X [1] is applied, which stipulates that memory is either writable or executable (but not both). Return-oriented programming (ROP) [25] is a code-reuse injection attack technique that exploits a software vulnerability by chaining existing *gadgets* (small snippets of code ending in a return opcode) together in arbitrary ways. The Advanced RISC Machine (ARM) and a highly customizable and configurable Tensilica Xtensa processors which are widely used in many Internet of Things (IoT) and embedded devices are now becoming a more appealing target of ROP attacks to acquire the capability to control a system's behavior. In general, injection attack has a broad application area that can be damaging even for applications that are not traditionally considered critical targets, such as personal websites, because attackers can use them as footholds to launch more complicated attacks.

In a code injection attack, an adversary crafts a malicious input that gets interpreted by the application as code rather than data or overflows the stack buffer to divert the control flow

of the system in case of the code-reuse injection attacks. To catch such injection exploits, researchers have proposed different approaches depending on the platforms where the injections are applied. In this thesis, we are considering the injection attacks detection on web applications and embedded applications running on ARM or Xtensa processors.

To detect the code injection exploits on web applications such as SQL injection and cross-site scripting (XSS), taint analysis is a typical technique that is mostly proposed. Taint analysis detects security vulnerabilities [157, 140], by analyzing how user input originated from untrusted sources flows throughout the application. As a result, it has attracted much attention from both the research community and industry. However, appropriate (precise and scalable) security analysis tools are required to correctly evaluate the security of web applications. Besides, depending on the size and complexity of the vulnerable applications static or dynamic analysis can be applied, and considering the existing dynamic taint analysis approaches for java-based applications, most of them have prohibitive runtime overheads: even the most performant can impose a slowdown of at least 10–20% and often far more [32, 17, 76, 45]. Although black-box fuzzers can be connected with taint tracking to detect vulnerabilities in the lab, it is difficult to use these approaches on stateful applications or those that require structured inputs [77, 58].

Similarly, to detect the injection attacks that exploit memory bugs on embedded applications (e.g., ROP and its sibling *jump-oriented programming* (JOP) [22]) some techniques have been proposed for the applications running on ARM. The techniques try to enforce control flow integrity (CFI) via dynamic binary instrumentation (DBI) [65, 112] or the ARM CoreSight debugger [83, 84, 85, 103, 102], supplemented with *meta-data* collected by static analysis. However, the techniques that use dynamic instrumentation suffer from high performance overhead while those that use the ARM CoreSight debugger suffer from high storage overhead. Besides, the hardware monitor that uses the hardware debugger could drop traces given a sufficiently high branch rate since the monitor requires more time to process a trace than the rate at which branches occur on the target processor [41]. Therefore investigating whether another line of defense can detect ROP attacks on ARM accurately and precisely with low performance and storage overhead is advisable. Similarly, although Xtensa is also vulnerable to ROP attacks, it has only been investigated rudimentary. Hence investigation of ROP attack detection mechanisms on this platform is also interesting since Xtensa is used in almost every firmware-based embedded WiFi home automation device.

In this thesis, we propose novel approaches to detect injection attacks on different platforms precisely with low performance and/or storage overhead. In general, we answer the following research questions: **RQ1**. How to precisely detect code injection attacks on web applications using taint analysis? **RQ2.** How HPC combined with machine learning techniques can detect injection (control flow) attacks on embedded applications running on

ARM or Xtensa processors? Our work comprises the following three published papers [168, 60, 104] and one submitted paper [170] each of which contributes to the precise detection of the injection attacks as presented in this thesis:

To answer the **RQ1**, we considered static analysis for security evaluation of a complex hybrid app (JavaScript and native) provided by an industrial partner, and dynamic taint tracking combined with existing tests for injection attacks detection on Java-based web applications.

Static Security Evaluation [168]: To precisely detect injection attacks on the hybrid industrial application, we initially compare the precision, scalability and code coverage of two widely-used JavaScript static analysis frameworks—WALA [121, 142] and SAFE [82, 116]. The reason behind the comparison is to choose the analysis tool that can precisely evaluate the security of the web application with tolerable scalability as security analysis mainly depends on the precision of the points-to analysis approach. Moreover, as the hybrid app is very complex with large code size, we choose static analysis to cover all possible flows and prevent the runtime overhead of the dynamic analysis. Since WALA and SAFE use different ways of intermediate representations, the comparison of the analyzers is not straightforward. Thus, we integrate WALA's analysis into SAFE to extract appropriate elements for the comparison. After that, we select objects and global variables of the user program that are registered for the same source location in both analyzers. To appropriately evaluate the precision of the analyzers, we first select all equivalent user objects and variable references. Then, we compute the average points-to set sizes over all fields (i.e., pointer keys) of the selected user objects and variable references. The precision evaluation using the hybrid app and other 12 benchmarks [74, 168] results in 1.073 and 1.93 average points-to set sizes for SAFE and WALA, respectively. We also evaluate their scalability and code coverage by measuring the analysis time and the number of object fields having non-empty points-to set size, respectively. For all benchmarks WALA requires less time to finish the analysis, indicating that it is relatively more scalable but it gives lower coverage. The evaluation indicates that SAFE provides higher precision and better code coverage at the cost of tolerable lower scalability. Hence, as the precision of the points-to analysis has a higher impact on the security evaluation, we choose SAFE for our security analysis of the hybrid app.

The hybrid industrial app is structured in the form of Node.js modules that reside in different paths of the project and cannot be analyzed by SAFE directly, as it requires all code to be present in one directory. Thus, we contribute *simplePack*, a source code transformation tool that bundles module dependencies in a way that is more suitable for static analysis. We compare *simplePack* to *Browserify* [5] by measuring the precision and recall of their bundled programs' static callgraph in WALA. The evaluation shows that *simplePack* displays better

precision and recall overall. The hybrid app, built on a middleware platform, includes more than 300 modules and contains more than 230,000 lines of code, which makes a direct whole program static analysis almost impossible. Hence we modeled the platform functions in SAFE and removed parts of the code not relevant to the main task of the app while keeping its major semantics and features.

Then, we extended the SAFE's taint analysis to detect code injection attacks due to any untrusted flow including objects containing tainted fields rather than only tainted primitive values. We also model the JSON.stringify function, which acts as an input sanitizer by changing the input value or object to a non-executable JSON string. Note that this function just acts as a prototypical sanitizer to evaluate whether our approach can support sanitization. Which function may act as a sanitizer depends heavily on the semantics of the sink. Security evaluation of four independently executed programs of the hybrid app using our extended SAFE's taint analysis approach detected untrusted (tainted) objects at the sink for the two tests. For the other two, the untrusted user input was not detected at the sink since the tainted value was sanitized before reaching the sink. In general, the evaluation of the hybrid app using the more precise points-to analysis of SAFE and its extended taint analysis manifests that there is a potential for injection attacks, as tainted objects may reach the sink without being sanitized. However, we could not exploit these vulnerabilities because the hybrid app uses a strong attack protection mechanism on the server-side. The detailed approach and result of our evaluation are published in our SAC'19 paper [168] which constitutes Chapter 3 of this thesis.

RIVULET [60]: Our key idea to detect injection attacks on the Java-based vulnerable applications is to use dynamic taint tracking before deployment to amplify developer-written tests to check for injection vulnerabilities. These integration tests typically perform functional checks. Our approach re-executes these existing test cases, mutating values that are controlled by users (e.g., parts of the test's HTTP requests) and detecting when these mutated values result in real attacks. To our knowledge, this is the first approach that combines dynamic analysis with existing tests to detect injection attacks.

Key to our test amplification approach is a white-box context-sensitive input generation strategy. For each user-controlled value, state-of-the-art testing tools generate hundreds of attack strings to test the application [109, 108, 77]. By leveraging the context of how that user-controlled value is used in security-sensitive parts of the application, we can trivially rule out most of the candidate attack strings for any given value, reducing the number of values to check by orders of magnitude. Our testing-based approach borrows ideas from both fuzzing and regression testing, and is language agnostic.

When applied to the version of Apache Struts exploited in the 2017 Equifax attack, Rivulet quickly identifies the vulnerability, leveraging only the tests that existed in Struts at that

time. We compared RIVULET to the state-of-the-art static vulnerability detector Julia on benchmarks, finding that RIVULET outperformed Julia in both false positives and false negatives. We also used Rivulet to detect new vulnerabilities. The detail of our work is well expressed in our ICSE 2020 paper [60] and it covers Chapter 4 of this thesis.

To answer the **RQ2**, we separately investigate the injection attack detection on ARM and Xtensa platforms.

Injection Attack Detection on ARM [170]: To detect ROP and JOP attacks on ARM, we combined the HPC and machine learning techniques. Note that the myriad of HPC events on ARM differs from x86, as well as the execution model (e.g., there is no dedicated return opcode on ARM). The HPCs count the occurrence of certain hardware events in the ARM processor when executing a program, but it has not been investigated whether the events for normal and ROP attack executions differ significantly enough to enable automatic detection. We create a machine learning model of the behavior on ARM-based Raspberry Pi machines to address this question empirically.

Our machine learning approach computes models for runtime monitoring. The *offline training* examines several machine learning techniques and generates a set of classification models from HPC training data collected during benign executions and attacks. On top of previous work, we developed a tracer that starts the recording of HPC events of executions with injected code only when the actual attack commences (i.e., the first gadget of the exploit executes), which improves the classifier's accuracy by 12%. The *runtime monitor* contains a *modified program loader*, a *kernel module* and a *classifier*. The program loader configures the CPU using the tool *perf* as to track the set of HPCs required for the trained classifier, the kernel module computes the delta of these HPCs each time an interrupt occurs and feeds these values to the machine learning-based classifier, which labels the recent program execution as an attack or benign.

To obtain an optimal classification model our approach trains models using multiple machine learning approaches. Of eight machine learning techniques examined, the optimal classification model of offline training – SVM with a RBF kernel – displays 92% and 91% accuracy for Raspberry Pi 4 and Pi 3, respectively. Leveraging this optimal classifier we evaluate ROP attack detection via runtime monitoring on Raspberry Pi using 15 exploits (based on four ROP attack variants) of real-world vulnerable applications. The detection of these attacks at runtime provides 75% accuracy, and we will elaborate on possible technical reasons for that difference. The detail of our ROP attack detection approach and evaluation result on the ARM platform has been submitted as a conference paper to COMPSAC2023 and constitutes Chapter 5 of this thesis.

Injection Attack Detection on Xtensa [104]: we present the first detection of ROP, and its variant *Jump-oriented programming* (JOP), in a firmware-only environment (based on the Xtensa platform) using *hardware performance counters* (HPCs). Our approach depends on the variations in the HPC micro-architectural events triggered by ROP and normal program execution. We implemented attack scenarios using instrumented programs and exploits that perform operations similar to those in known microprocessor benchmark programs. Recorded micro-architectural events were used to train machine learning binary classifiers. The learned model identifies relevant HPCs, which could serve as predictors of ROP/JOP execution even in configurations where features non-typical to conventional processors, like instruction memory and data memory, are available. Our evaluation results also indicate a high precision, recall, and accuracy of the classifier predictions. The result of our investigation is published in SAC'22 [104] and is presented in Chapter 6 of this thesis.

Last but not least, this doctoral thesis deals with injection attack detection on web applications using taint analysis and detection of ROP and JOP attacks combining HPC and machine learning on ARM and Xtensa processors. We choose SAFE which provides a precise points-to-analysis approach to evaluate the security of a JavaScript-based industrial hybrid app by extending its taint analysis. For the detection of Java-based applications, we provide *RIVULET*, which combines the power of human developers' test suites and automated dynamic taint analysis. Similarly, for applications running on ARM and Xtensa platforms, we investigated the detection and prevention of ROP and JOP attacks by combining HPC and machine learning. So, this thesis shows how relevant taint analysis is for injection attack detection on web applications and the power of HPC combined with machine learning in the control flow injection detection on ARM and Xtensa platforms.

1.1 Thesis contributions

This section summarizes the major contributions of this thesis as follows:

1. Security evaluation of an industrial hybrid app from a partner company using static taint analysis which contains the following sub-contributions.

    - A comparison of JavaScript static analyzers (WALA's and SAFE's) points-to analysis using equivalent user-defined objects.
    - A framework-agnostic module bundler suitable for static analysis, which we compare against the off-the-shelf bundler Browserify.
    - Security analysis of the hybrid industrial app by extending the SAFE's taint analysis to support tainted complex objects rather than tainted primitive values.

2. Revealing injection vulnerabilities by leveraging existing tests for java based web applications using dynamic taint analysis. This covers the following sub-contributions.

    - A technique for re-using functional test cases to detect security vulnerabilities in java based applications using dynamic analysis
    - Context-sensitive mutational input generators for SQL, OGNL, and XSS that handle complex, stateful applications
    - Embedded attack detectors to verify whether rerunning a test with new inputs leads to valid attacks.

3. Detecting and preventing ROP attacks using machine Learning and HPC on ARM platforms covering the following sub-contributions.

    - Control-flow attack detection on the ARM platform using HPCs and machine learning techniques consisting of offline training and runtime monitoring.
    - A debugger (tracer) that selectively records the actual attack section of a program subject to a control flow attack to improve the classification model.
    - Creation of a benchmark of 15 exploits (of four ROP variants) for the ARM platform (i.e., Raspberry Pi) from 8 real-world vulnerable applications leveraged for the offline training and online monitoring.
    - An evaluation of eight machine learning techniques to find the optimal classification model for ROP attack detection on ARM platform.

4. Detecting ROP attacks on firmware-only embedded devices using machine learning and HPCs. This includes the following sub-contributions.

    - Show how ROP and JOP attacks could be orchestrated on Xtensa processors
    - Present the first practical work on detecting ROP and JOP attacks in a firmware-only embedded system using HPC and Machine learning.

1.2 Publications

This thesis covers the following previously published papers in proceedings of peer-reviewed international conferences:

1. **Gebrehiwet B. Welearegai**, Max Schlueter, and Christian Hammer. *Static security evaluation of an industrial web application*. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pages 1952–1961, 2019.

   In this paper we propose a novel approach to compare the precision, scalability and code coverage of two widely-used static analysis frameworks—WALA and SAFE—together with simplePack, which analyzer-agnostically bundles dependent modules, enabling a fair comparison. Then we analyze the data flows of a hybrid app (from a partner company) by extending SAFE's taint analysis. This paper forms Chapter 3 of the thesis.

2. Katherine Hough, **Gebrehiwet Welearegai**, Christian Hammer, and Jonathan Bell. *Revealing injection vulnerabilities by leveraging existing tests*. In Proceedings of the International Conference on Software Engineering(ICSE), 2020.

   This paper describes a new approach for detecting injection vulnerabilities in applications by harnessing the combined power of human developers' test suites and automated dynamic analysis. Chapter 4 provides more detail on this work.

3. **Gebrehiwet B. Welearegai**, Chenpo Hu, and Christian Hammer. *Detecting and Preventing ROP Attacks using Machine Learning on ARM*. **Submitted** to 47th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2023

   In this paper, we investigate whether ROP attack detection and prevention based on hardware performance counters (HPC) and machine learning can be effectively transferred to the ARM architecture. The detail of the this paper is presented in Chapter 5 of this thesis.

4. Adebayo Omotosho, **Gebrehiwet B. Welearegai**, and Christian Hammer. *Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters*. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pages 510–1519, 2022

   we present the first detection of ROP, and its variant Jump-oriented programming (JOP), in a Xtensa based firmware-only environment using hardware performance counters (HPCs). This paper forms Chapter 6 of the thesis.

Publication not covered in this thesis:

1. **Gebrehiwet B. Welearegai** and Christian Hammer. *Idea: Optimized automatic sanitizer placement.* Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings 9. Springer International Publishing, 2017.

   This paper is an extension of my master's thesis which is published while doing my PhD study. The paper presents an optimized automatic sanitizer placement to support the developers who face the burden of sanitizer placement.

1.3 Thesis Outline

The rest of this thesis is structured as follows: Chapter 2 provides definitions of the concepts introduced above and lays the foundation for the remaining chapters. In particular, injection attack approaches (XSS, SQL, ROP, JOP) and the platforms (Web, ARM, Xtensa) where the analyses are performed. Moreover, different analysis techniques such as pointer analysis, taint analysis, and machine learning are explained.

The main body is composed of Chapters 3, 4 and 6 which are the result of published research and Chapter 5 which contains the contents of a submitted research work. At the high-level Chapter 3 and 4 mainly focuses on the detection of injection attacks (such as XSS and SQL) on web applications whereas Chapter 5 and 6 focus on the detection of control flow injection attacks (such as ROP and JOP) on embedded applications running on ARM and Xtensa processors, respectively.

In particular, Chapter 3 presents the evaluation of JavaScript static analysis frameworks and the security evaluation of a real-world hybrid application using the selected precise static analysis framework during the comparison of the tools. Chapter 4 presents an injection attack detection approach combining the power of human developers' test suites and automated dynamic analysis.

Chapter 5 examines how to detect control flow injection attacks (ROP and JOP) on ARM processors using hardware performance counters (HPCs) and machine learning. Online monitoring based on an optimal model generated by offline training is used to detect the control flow injection attacks. Chapter 6 also presents the ROP and JOP attack detection using HPC and machine learning but now focuses on Firmware-Only Embedded Device Using Xtensa processors.

Last but not least, Chapter 7 presents the related works of injection attack detection on different platforms. Chapter 8 and 9 provides the conclusion and future research direction of this thesis, respectively.

Chapter 2

Background

2.1 Injection Attacks

Injection vulnerabilities allow an attacker to supply untrusted input to a program, which gets processed by an interpreter as part of a command or query which alters the course of execution of that program. Injection attacks come in a variety of flavors, as attackers may be able to insert different kinds of code into the target application.

SQL Injection (SQLI) is the most classic type of injection attack where attackers can control the contents of an SQL statement. For instance, consider this Java-like code snippet that is intended to select and then display the details of a user from a database: `execQuery ("SELECT * from Users where name = '" + name + "'");`. If an attacker can arbitrarily control the value of the `name` variable, then they may perform a SQL injection attack. For instance, the attacker could supply the value `name = "Bob' OR '1'='1"` which, when joined to the query string will produce `where name = 'Bob' OR '1'='1 '`, which would result in *all* rows in this user table being selected. SQLI attacks may result in data breaches, denial of service attacks, and privilege escalations.

Remote Code Execution (RCE) attacks are a form of injection attacks where an attacker can execute arbitrary code on an application server using the same system-level privileges as the application itself. Command injection attacks are a particularly dangerous form of RCE where an attacker may directly execute shell commands on the server. Other RCE attacks may target features of the application runtime that parse and execute code in other languages such as J2EE EL [106] or OGNL [151, 152].

Cross-site Scripting (XSS) attacks are similar to RCE, but result in code being executed by a user's browser, rather than on the server. XSS attacks occur when a user can craft a request that inserts arbitrary HTML, JavaScript code, or both into the response returned by the server-side application. Such an attack might hijack a user's session (allowing the attacker to impersonate the user on that website), steal sensitive data, or inject key loggers. Server-side XSS attacks may be *reflected* or *persistent*. Reflected XSS attacks are typically used

in the context of a phishing scheme, where a user is sent a link to a trusted website with the attack embedded in the link. Persistent XSS attacks occur when a payload is stored in the host system (*e.g.*, in a database) and is presented to users who visit the compromised site.

Code reuse attacks are another form of injection attacks that exploit memory bugs by maliciously altering the program's behavior in applications written in low-level languages like C or C++. The code reuse attack can even take full control over the control-flow of the applications. The memory exploitation can be done directly by injecting a new machine code into the vulnerable program's memory. However, direct injection is not mostly possible since a protection technique known as W $\bigoplus$ X [1], which ensures that memory is either writable or executable (but not both), is applied. Hence, attackers can use existing code in the code or libc section of the stack to indirectly inject code and divert the program's control flow. Return-into-libc (RILC) [133] is a relatively simple code-reuse attack in which an adversary uses a buffer overflow to overwrite part of the stack with return addresses and parameters for a list of functions within libc. By doing so control is sent to the beginning of an existing libc function, such as *system()*. To achieve greater expressiveness, return-oriented programming (ROP) [25] was introduced to exploit software vulnerability by chaining existing gadgets (small snippets of code ending in ret) together in arbitrary ways. Section 2.7 presents a detailed explanation of how ROP attack works on ARM platforms.

2.2 Points-to Analysis

A points-to analysis is a static code analysis that attempts to determine the possible values of a pointer or heap reference in a program. There are several pointer analysis algorithms, but the most common pointer analysis algorithms are Andersen [7] and Steensgaard [145]. The Andersen-style pointer analysis is expressed as subset constraints. In contrast, Steensgaard-style pointer analysis uses equality constraints. Specifically, it treats every input program statement as an indication that some points-to-sets should be unified, i.e., become one [136]. The Andersen-style analysis is slower but more precise than Steensgaard-style. Table 2.1 illustrates the constraints of both algorithms. In the table entries loc(x) and pts(x) indicate the location (address) of x and points-to-set of x, respectively.

TABLE 2.1: Andersen and Steensgaard constraints

Constraint Type	Assignment	Andersen		Steensgaard	
		Constraint	Meaning	Constraint	Meaning
Base	a = &b	a $\supseteq$ {b}	loc(b)$\in$pts(a)	a= {b}	loc(b)$\in$pts(a)
Simple	a=b	a$\supseteq$b	pts(a)$\supseteq$pts(b)	a=b	pts(a)=pts(b)
Complex	a = *b	a$\supseteq$*b	$\forall$v$\in$pts(b). pts(a)$\supseteq$ pts(v)	a = *b	$\forall$v$\in$pts(b). pts(a)= pts(v)
Complex	*a = b	*a$\supseteq$b	$\forall$v$\in$pts(a). pts(v)$\supseteq$ pts(b)	a = *b	$\forall$v$\in$pts(a). pts(v)= pts(b)

```
1 //url==https://...#name=<script>alert("xss")</script>
2 var url = document.URL
3 var pos = url.indexOf("name=")+5
4 document.write(url.substring(pos, url.length))
5
```

FIGURE 2.1: Taint analysis example

A points-to analysis algorithm can be either flow-sensitive or flow-insensitive. The **flow-sensitive** analysis takes into account the order in which the statements in the program may be executed. The program is handled as a sequence of statements and the memory location where pointer expression may refer to is computed for each statement, i.e., it is statement level analysis. On the other hand, the **flow-insensitive** analysis does not take into account the order rather the program is handled as a set of statements. Hence, it computes the memory locations where pointer expressions may refer to, at any time in the program execution, i.e., it is program level analysis.

2.3 Taint Analysis

Taint analysis emerged as a useful technique for discovering security vulnerabilities in web applications [157]. Security taint analysis is an information-flow analysis that automatically detects flows of untrusted user input to security-sensitive computations (integrity violations) or flows of private information into computations that expose it to public observers (confidentiality violations) [140]. The starting point of taint analysis is typically a data source API call. For instance, consider the code sample in Figure 2.1 which is vulnerable to a DOM-based XSS attack. The variable *name* could contain malicious JavaScript code in a script tag that is then executed by `document.write`. Considering `document.URL` as a source and `document.write` as a sink, the taint analysis raises an alarm that untrusted user input is flowing into a sink. An application developer can endorse untrusted inputs via sanitization[1] functions, which the taint analysis needs to consider, in order to reduce the number of false positives.

2.4 JavaScript Static Analysis Frameworks

Static analysis mechanisms automatically derive certain properties from a program without executing it. Currently, there are some frameworks that analyze JavaScript[2] applications statically such as TAJS [70], JSAI [74], WALA [121], SAFE [82, 116]. However, reviewing the scientific literature WALA and SAFE are the most commonly used frameworks.

[1] A set of transformation functions that render attacks harmless, such as: "escape all quotation marks in user input".

[2] JavaScript analyses usually support a subset of a standard called ECMAScript (ES), ES5 is now common, ES6 still new.

WALA provides soundy [90] flow-insensitive static analysis for both Java bytecode and JavaScript. However, JavaScript's ability to create and delete properties at runtime presents a great challenge for scalable and precise points-to analysis. Although WALA provides correlation tracking [142], improving points-to analysis scalability and precision via smart handling of *for-in* loops, scalability remains a problem for *inter-procedural, finite, distributive subset* (IFDS)-based analyzers. Therefore, WALA intentionally introduced a new unsound but more scalable static analysis that constructs a field-based (FB) call graph [48], i.e., uses one abstraction for all instances of each property in the whole program. To further improve the pointer analysis scalability and to eliminate calls to eval, [128] proposed a dynamic analysis. However, the coverage of the dynamic analysis may not be sufficient as the analysis observes only one program execution at a time.

SAFE [82] is a flow- and context-sensitive static analysis framework that provides both formal specification and its open-source implementation for JavaScript. Analysis scalability is greatly improved by using loop sensitivity (LSA) [114] that handles loops more precisely, which turns out to be a determining factor in terms of analysis precision as well. It also supports the *with* statement [113], rewriting it to semantically equivalent code when it does not contain any dynamically generated code. Recently, SAFE 2.0 [116] supports *pluggability* (ability to select analysis techniques at runtime), *extensibility* (APIs for adding new phases) and *debuggability* (HTML and console debugging), improving user-friendliness. In contrast WALA's source code repository is huge and complex. SAFE is able to analyze most ES5[3]codes precisely. To analyze object properties more precisely, SAFE uses *recency abstraction* [16, 115] which performs strong updates on recently allocated objects and weak updates on joined *old* objects.

2.5 Node.js Platform

Node.js is an open-source, cross-platform JavaScript runtime environment, built on Chrome's V8 JavaScript engine, for executing JavaScript server-side code [35]. It uses an event-driven, non-blocking (asynchronous) I/O model that makes it lightweight and efficient. The node package manager (*npm*) is used to install modules and manage code dependencies from the command line. Node.js has built-in and user-defined JavaScript modules. Modules structure a program into separate sub-programs to simplify the development and maintenance of complex applications. Each module has its own scope and cannot pollute the namespace of other modules.

Figure 2.2 illustrates how modules are used in Node.js. Module *A.js* exports a function constructor and is required by the main module `main.js`. In main.js we instantiate a new object from `A` and call the method `foo` on it. The `require` function has diverse semantics,

[3]However, not all of ES5 is currently supported (e.g. getter/setters are not modeled).

```
1 // main.js
2 var A = require("./A.js");
3 var a = new A();
4 a.foo(2) // => false
5
```

(A) main module

```
1 // A.js
2 function A() {
3   this.foo = function (x) {
4     return x === 0; };
5 }
6 module.exports = A;
7
```

(B) dependent module

FIGURE 2.2: Node.js program example consisting of two modules that reside in the same directory.

e.g., loading built-in modules in Node.js or recursively searching through directories for modules installed by npm.

2.6 Industrial Hybrid App Setup

For our security analysis and evaluation of the static analyzers, we use a real-world industry application which is structured into Node.js modules. This section provides an overview of a practical Node.js based hybrid web app developed by ApproLogic GmbH that is used for our static analysis. The application is structured into Node.js modules and makes calls to middleware platform functions that are attached to the module scope during execution. Figure 2.3 shows an overview of the module dependencies in the hybrid app from the *LoginController* slice[4]. The *Platform* module represents the middleware platform and consists of several modules. The hybrid app includes more than 300 modules and contains more than 230,000 lines of code. The platform functions perform many interactions with built-in library functions such as *jQuery*, *Lodash* and *Backbone.js*.

2.7 Return-Oriented Programming on ARM

As explained in Section 2.1, the ROP attack is a code-reuse attack in which an adversary generally leverages a buffer overflow to overwrite parts of the stack in order to divert the program's control flow to existing executable code sections of the program.

The core idea of ROP on ARM is to exploit the presence of gadgets (small instruction sequences) that induce some well-defined behavior, such as returning using *pop* or branching using *blx* instructions [39]). Figure 2.4 presents two gadgets, one ending in *pop* and another

[4]The other controllers are: *RegisterController*, *SigninController* and *AssistanceController*

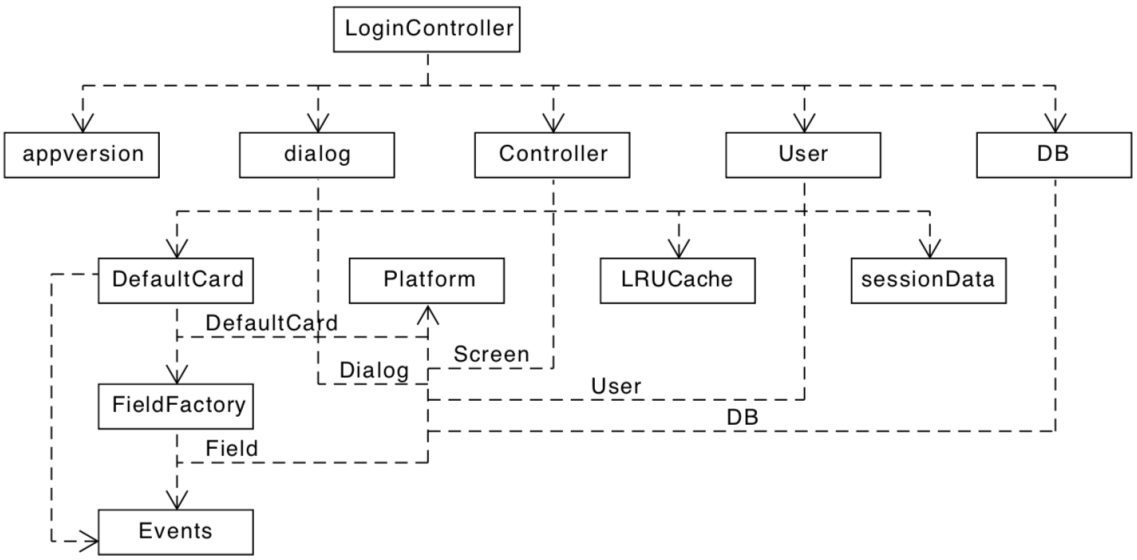


FIGURE 2.3: Module dependence graph of the ApproLogic hybrid app. The nodes represent modules and the dashed edges represent require dependencies.

```
1    mov r0, r4;
2    pop {r4, pc};
3
```

(A) pop based gadget

```
1 mov r0, r7 ;
2 blx r3;
3
```

(B) branch based gadget

FIGURE 2.4: Gadget examples in ARM platform

ending in *blx*. In the first gadget the first instruction moves the value in register *r4* to register *r0* (which is used as the first argument of a function call), subsequently the values on the top of the stack are popped to registers *r4* and *pc*, which alters control flow depending on the value loaded to *pc*. Similarly, the instruction *blx r3* in the second gadget changes the flow of the program to the address in *r3*. Selecting such gadgets (e.g., with the help of gadget discovery tools such as ROPgadget[5]) and chaining them together properly, an adversary can build complex exploits to induce arbitrary behavior in the target program to a malicious end.

2.7.1 ROP Variants on ARM

There are several ROP variants on ARM but we explain here the most common ROP variants which are used in this thesis to increase the diversity of the training and testing set of the machine learning techniques.

Return-to-Zero-Protection (Ret2ZP) Ret2ZP[14, 66] attack allows an adversary to control

[5]https://github.com/JonathanSalwan/ROPgadget

the argument registers r0-r3 from the stack and direct the control flow (CF) into a library function. This attack is similar to the Ret2LibC [174] attack on the x86 system which overwrites a ret-address on the stack with the *LibC* function address to perform the attack. The Ret2ZP attack first of all places the *LibC* function arguments and the LibC function address onto the stack. Then the CF is redirected to the vulnerable code sequence (VCS) which moves the arguments from the stack to the argument registers, and moves the LibC function address from the stack to the PC, redirecting CF to the LibC function.

Return-to-mprotect (Ret2mP) mprotect() is a function which takes 3 arguments to set the access rights (READ—WRITE—EXECUTE) to a memory area. The arguments are

1. A starting address of the memory area to change
2. Length of the memory area to change
3. The new mask of permissions for that memory area (READ—WRITE—EXECUTE is 0x07)

Stack Pivoting With stack pivoting, an adversary can create a fake stack in another memory area (such as in the heap), so that the adversary can take control of the code and command execution. E.g., the stack can be pivoted by controlling data pointed to by SP, so that each ret-instruction results in increasing SP and transferring execution to the next gadget address chained by the adversary. In order to enrich the diversity of the data gathered for machine learning, a ROP attack with stack pivoting on a real-life application is also implemented in this thesis.

Jump-oriented programming (JOP) According to the differences in the (control-flow manipulating) last instruction, ROP attacks can be classified into *ret-based* ROP and *jmp-based* ROP or jump-oriented programming (JOP). Since ARM does not provide a *ret* opcode, the *pop* instruction, which moves a return address from the stack into the pc is used instead, i.e., gadgets ending in *pop(...,pc)* can be used to perform a ROP attack. Conversely, JOP uses gadgets ending in a *blx r* instruction, where *r* represents a general-purpose register that stores a gadget's address. Figure 2.4 contains *pop*-based and branch (*blx*) based gadgets for ROP and JOP attack variants, respectively. The other attacks explained above (i.e., Ret2ZP, Ret2mP and Stack Pivoting) can also be generalized into ROP and/or JOP attacks based on the gadgets they used to create the attack exploit. The comparison of ROP and JOP attacks on the x86 platform was initially presented by Bletsch et al. [23]. For the ARM platform, we have designed our own model, which is presented in Section 5.3.1

2.8 Xtensa Architecture and Registers

The Xtensa processor architecture is a Harvard architecture with instruction and data memory separate that provide fast simultaneous access to both memories. The Xtensa processor

TABLE 2.2: Registers in Call0 and Windowed Register ABI

Registers	Call0 ABI	Windowed Register ABI
a0	Return address	Return address
a1	0 or (sp) Stack Pointer (callee-saved)	Stack pointer
a2 - a7	Function Arguments	Incoming arguments
a7		Callee's stack-frame pointer (optional)
a12 - a15	Callee-saved	
a15	Stack-Frame Pointer (optional)	

architecture targets embedded system-on-a-chip applications, and the *Instruction Set Architecture* (ISA) specifies a 32-bit RISC-like architecture expressly designed for embedded applications. The Xtensa core ISA is implemented as 24-bit instructions, providing about a 25% reduction in code size compared with 32-bit ISAs [27]. Instructions can be represented as 16 or 24 bits, which results in high code density and also means that any byte is a valid jump target. The instructions provide access to the entire processor hardware and support special functions, such as a single-instruction *compare and branch*, which reduces the number of instructions required to implement various applications. Xtensa has three distinguishing features and the first is *extensibility*. This addition of architectural enhancements allows easy and efficient extension of the processor architecture with application-specific instructions. The second is *configurability*, which supports creating custom processor configurations that make it easy to specify whether (or how much) pre-designed functionality is required for a particular product. The third is *retargetability*, which allows mapping of the architecture onto hardware to meet the different speed, area, and power targets in different processes. These features make Xtensa unique and in demand for embedded systems design. Xtensa supports 16 address registers a0 to a15, where the functionality of these registers differs slightly depending on the *application binary interface* (ABI) in use. An ABI is a set of rules describing what happens when a function is being invoked, how its parameters are processed, and defining the stack layout for the function call. Xtensa supports two ABIs: *Call0* ABI and the *Windowed Register* ABI, for which Table 2.2 presents the registers and their functions. The Call0 ABI works with all Xtensa processors, and it has a better context switch time than the Windowed Register ABI. ROP and JOP attack orchestration on both ABIs are similar [86] despite the differences in register usage.

The Xtensa architecture also has a 32-bit program counter, which – similar to x86 and in contrast to ARM – cannot be directly accessed. Generally, Xtensa's instruction format follows the pattern:

mnemonic <*dest_reg* >, <*operand_1*>,<*operand_2*>

The destination register *dest_reg* stores the result of the operation specified by the opcode

mnemonic on the first *operand_1* and second *operand_2* operand. Xtensa's instruction set is very flexible and not all of the instruction set requires all of the fields in this template. More specific details about Xtensa LX hardware instruction set architecture can be found in [28].

2.9 Hardware Performance Counters (HPCs)

HPCs, which have been available in modern processors (such as ARM, AMD, and Intel) for more than a decade [38], monitor and measure events that occur at the CPU level during process execution related to instructions (e.g., cycles, instruction retired), memory accesses (e.g., cache misses or hits, main memory misses or hits), etc. Intel introduced the performance monitoring unit (PMU) [37] which supports the monitoring of two kinds of HPCs, counters of architectural hardware performance events and counters of non-architectural hardware performance events. Architectural hardware performance events are the same in different processors. However, the non-architectural hardware performance events might differ based on the processor, i.e., they are specific to the micro-architecture, such as branch prediction, cache and translation look-aside buffer (TLB).

In order to obtain HPC information, initially the HPCs must be configured according to the events of interest. Then, polling or sampling can be used to read the HPC values at runtime [38]. When polling is used the HPCs can be read at any instant whereas for event-based sampling the occurrence of events triggers reading HPCs. The event-based sampling is enabled through Performance Monitoring Interrupt (PMI) in most CPUs. The PMI can be generated after a certain number of events are occurred [38]. For instance, the HPC can be configured with a certain threshold. Then once a counter exceeds that threshold, it will result in the generation of a PMI. At each PMI, the numbers of arithmetic, cycles, call and return instructions are read [38].

Though the initial purpose of HPCs was for debugging, they have also been used in several other applications, such as detecting program modification at a low cost [165, 91] and vulnerability research [117]. They have also been used extensively in non-embedded processors for malware detection with good detection accuracy [177, 3, 92].

Profiling tools, such as Linux's perf[6], allow HPC data to be obtained using several methods, but that flexibility comes at the expense of yielding different counter values for the same application due to the multi-process environment and the non-determinism of HPCs [38].

There are no HPC standards and they are therefore manufacturer dependent; on a different microprocessor, even with different models of the same processor family, HPCs may have different names, numbers, and functions. Even though modern processors can record a large number of events at a time, only a fraction of these events can be monitored at a time. The number of events that can be monitored is determined by the number of available

[6] https://en.wikipedia.org/wiki/Perf_(Linux)

HPCs which is usually fewer than the overall number of possible events. For example, both the ARM and Xtensa processors we used in this thesis can monitor only 8 HPC events simultaneously. .

2.10 Machine Learning

Machine learning is a collection of methods that enable computers to automate data-driven model building and programming through a systematic discovery of statistically significant patterns in the available data [20]. Machine learning algorithms build a mathematical model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so. Machine learning algorithms are used in a wide variety of applications, such as email filtering, Malware detection and other similar tasks which can not be easily solved using conventional algorithms.

Machine learning algorithms are often categorized as supervised or unsupervised. In supervised learning input examples and their desired outputs are given to learn a general rule that maps inputs to outputs, i.e., the system is able to provide targets for any new input after sufficient training. Whereas in unsupervised learning, no labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be used to discover hidden patterns in data or it can be a means to feature learning.

2.10.1 Types of Machine Learning Methods

The supervised learning supports various algorithm types

- *K neighbors:* It is an algorithm that stores all available cases and classifies new cases by a majority vote of k neighbors.
- *Boosting:* a family of machine learning algorithms that convert weak learners to strong ones.
- *Decision trees:* uses a tree-like model of decisions and their possible consequences. Decision trees often perform well on imbalanced data sets because their hierarchical structure allows them to learn signals from both classes.
- *Naïve Bayes:* It is a classification technique based on Bayes' theorem with an assumption of independence between predictors, i.e., the presence of a particular feature in a class is unrelated to the presence of any other feature. It is frequently used in imbalanced dataset problems.
- *Support Vector Machine (SVM):* constructs a hyperplane or set of hyperplanes for classification, regression, or other tasks. In handling a binary classification task, SVM is one of the methods reported to give a high accuracy in predictive modeling compared to the other techniques such as Discriminant Analysis [161].

Chapter 3

Static Security Evaluation of an Industrial Web Application

3.1 Overview

JavaScript is the most widely-used programming language for client-side web applications, powering over 95% of today's websites[1]. In spite of its popularity, it also introduces various errors, vulnerabilities, and ample challenges for program analysis: large-scale libraries, asynchronous event flows via user inputs, interaction between iframes, and new analysis domains like MVC frameworks and hybrid applications [148].

Researchers have proposed several static program analysis techniques to help JavaScript developers overcome some difficulties. Static analysis is challenging and imprecise due to dynamic language constructs like run-time code generation and heavy use of first-class functions [122]. Widely used static analysis frameworks like WALA [121, 142] and SAFE [82, 114, 116] represent the state of the art, however, little research assesses their strengths and weaknesses. [80] report the number of callees per call site of these frameworks but concentrate on the scalability gain of combining WALA and SAFE rather than their comparison. A recently released major rewrite of SAFE that aims at a more pluggable, extensible and debuggable framework (SAFE 2.0 [116][2]) raises the question which framework to leverage for static security analysis (including injection attack detection), which was the motivation for a thorough comparative analysis of WALA and SAFE.

Recently JavaScript's popularity also rose for server-side and desktop applications due to Node.js, an open-source, cross-platform runtime environment for executing JavaScript on the server-side[3]. Node.js programs are structured into modules, and can also be used in browsers by bundling up the module dependencies using *Browserify* [5], *Webpack* and *CommonJS Everywhere*. All these bundlers follow a similar pattern in how they package

[1] `https://w3techs.com/technologies/details/cp-javascript/all/all`
[2] From now onwards SAFE represents the version SAFE 2.0.
[3] `https://en.wikipedia.org/wiki/Node.js`

modules, but none of these are suitable for static analysis. Recently WALA and SAFE included direct support for Node.js modules, but their solutions are analysis-specific and thus not suitable for comparison. For example, the model in WALA is relatively complete (except for the native *core* modules), but SAFE's very preliminary. Hence, it is indispensable to have an analysis-agnostic bundler suitable for any static analysis framework for a fair comparison.

JavaScript- and Node.js-based applications can be vulnerable to various injection attacks [144]. Thus, security analyses like a taint analysis [157] are indispensable to detect potential injection vulnerabilities. The state of the art reports dynamic, static and hybrid taint analysis approaches [166, 126]. However, none of them are implemented in SAFE, which we selected for our security analysis after comparing the precision and scalability trade-off with WALA. SAFE supports taint analysis via a prototype implementation [123], but only reports tainted primitive type arguments such as strings that reach a sensitive sink, but not object type arguments, which our industrial hybrid app makes use of. Hence, it requires improving the SAFE's taint analysis to precisely detect potential injection vulnerabilities of the industrial app.

In this thesis, we present a thorough comparison of SAFE and WALA, introduce *simplePack* to bundle module dependencies suitably for static analysis, extend the taint analysis in SAFE, and analyze the security of a real-world industrial app using extended SAFE's taint analysis.

To extract appropriate elements for comparison of the analyses, we integrate WALA's analysis into SAFE and select objects and global variables of the user program that are registered for the same source locations in both analyzers. We compute the average points-to set sizes over all object fields (i.e., *pointer keys*) to evaluate the precision of both analyzers. We also evaluate their scalability and code coverage by measuring the analysis time and number of non-empty object fields (i.e., fields determined to point to an allocation site, e.g., for function objects), respectively. Our evaluation illustrates that SAFE provides higher precision and code coverage than WALA, but is less scalable. However, SAFE's lower scalability usually does not outweigh the gains in precision and code coverage. Our evaluation also shows that SAFE covers more code (determines receivers of function calls), which may also be responsible for the higher runtimes. Hence, we choose SAFE for our security analysis of the industrial app.

The hybrid industrial app is structured in the form of Node.js modules that reside in different paths of the project and cannot be analyzed by SAFE directly, as it requires all code to be present in one directory. Thus, we contribute *simplePack*, a source code transformation tool that bundles module dependencies in a way that is more suitable for static analysis. We compare *simplePack* to *Browserify* by measuring the precision and recall of their bundled

programs' static callgraph in WALA. The evaluation shows that *simplePack* displays better precision and recall overall. The hybrid app, built on a middleware platform, includes more than 300 modules and contains more than 230,000 lines of code, which makes a direct whole program static analysis almost impossible. Hence we modeled the platform functions in SAFE and remove parts of the code not relevant to the main task of the app while keeping its major semantics and features.

We analyze the security of the hybrid app by extending SAFE's taint analysis to identify tainted *objects* flowing to sinks. We also model the *JSON.stringify* function, which acts as an input sanitizer by changing the input value or object to a non-executable JSON string. Note that this function just acts as a prototypical sanitizer to evaluate whether we can support sanitization in our analysis. Which function may act as a sanitizer depends heavily on the semantics of the sink and is beyond the scope of this research. We evaluate the analysis on four components of the hybrid app and our taint analysis identifies the tainted parameter object due to the existence of tainted primitive property in the objects' property hierarchy. In contrast, tainted values passed through the *JSON.stringify* function before reaching a sink are correctly not reported as an illegal data flow.

3.2 Contributions

The major contributions of this thesis presented in this chapter are:

- A comparison of WALA's and SAFE's points-to analysis using equivalent user-defined objects.
- A framework-agnostic module bundler suitable for static analysis, which we compare against the off-the-shelf bundler Browserify.
- A static security analysis for a hybrid industrial app. To that end we model native platform functions and the sanitizer JSON.stringify directly in SAFE, and extended the SAFE's taint analysis to support tainted complex objects rather than tainted primitive values.

3.3 Motivating Example

The context- and loop-sensitivity, and the coverage of modeled functions have a magnificent effect on the precision of static analysis frameworks. The context-sensitive analysis distinguishes the different calling paths of a procedure expressed using *k-CFA* (Control Flow Analysis) such that *k* represents the depth of distinguished call hierarchies. Similarly, the loop-sensitive analysis distinguishes each iteration of loops with determinate loop conditions by using loop contexts expressed as *k-LSA* (Loop Sensitive Analysis) where *k* indicates the number of distinguished iterations in the analysis. The three programs in Figure 3.1 and their analysis results in Table 3.1 illustrate the effect of context- and loop-sensitivity as well as the model coverage on the precision of the analysis. For the context-sensitivity example

```
1 function f1(v) {
2   return f2(v);
3 };
4 function f2(v) {
5   return v; }
6 var o1 = {};
7 var o2 = {};
8 var x = {};
9 x.p = f(o1);
10 x.q = f(o2);
11
```

(A) context-sensitivity

```
1 var x={};
2 for (var i=0;i<3;i++){
3   x[i]=i;
4 }
5
```

(B) loop-sensitivity

```
1 var TopString = (45).toString
    (12);
2 var TopNum = Date.now()\right)
3
```

(C) model-functions

FIGURE 3.1: Examples to illustrate context-sensitivity, loop sensitivity and modeled functions

TABLE 3.1: Points-to analysis result of program examples in Figure 3.1. Columns with *keys* and *avgPts* headings represent the number of pointer keys and average points-to set size respectively.

program examples	1-CFA and 0-LSA				2-CFA, 3-LSA	
	SAFE		WALA		SAFE	
	keys	**avgPts**	**keys**	**avgPts**	**keys**	**avgPts**
a.	21	1.19	17	1.18	19	1.0
b.	2	1.0	3	2.67	5	1.0
c.	5	1.0	3	1.33	5	1.0

(Figure 3.1a), the number of pointer keys and average points-to set size in SAFE using *2-CFA* is reduced (i.e., points-to analysis become more precise) since the redundant old allocation sites, which are created due to weak updates[4], are minimized when context-sensitivity is increased. Particularly, using *1-CFA*, the old allocation site of argument v in function f_2 points to objects o_1 and o_2. But if we use *2-CFA* or more the old allocation does not exist and the recent allocation points-to o_2. Therefore, increasing the context-sensitivity leads to more precise points-to analysis as the weak updates are the main source of imprecision. However, WALA does not support context-sensitivity greater than *1-CFA* for JavaScript. Similarly, loop-sensitivity, which is not supported in WALA except for correlation tracking for *for-in* loops [142], is supported in SAFE and increases the precision by differentiating the pointers that depend on the iteration number. For instance, the *x[i]* pointer in Figure 3.1b varies and points to a different value at each iteration when *3-LSA* or more is used. Finally, the difference in the number of pointer keys for the example in Figure 3.1c is due to the fact that implicit primitive type conversion is modeled in SAFE but not in WALA.

[4]SAFE uses *recency abstraction* which performs strong updates on recently allocated objects and weak updates on joined *old* objects.

3.4 Methodology

In this section, we describe how the average points-to set is computed for programs analyzed using SAFE or WALA. Moreover, we explain how the SAFE's taint analysis is extended to detect tainted objects reaching the sinks and how *simplePack* bundles Node.js modules into a single bundled JavaScript file.

3.4.1 Average Points-to Set Computation

This section presents the general initial setup and the average points-to set computation approach of both static analyzers, i.e., SAFE and WALA.

Initial Setup

WALA uses a flow-insensitive Andersen-style pointer analysis approach to compute the points-to set of each pointer or heap reference for the whole program. In contrast, SAFE supports flow-sensitive points-to analysis and the heap state is different from statement to statement. Hence, we take the heap status at the *exit statement* of the top-level function as the state to compare the average points-to sets.

To have a more appropriate comparison, we integrate the WALA analysis result into SAFE following the approach of [80], i.e., we added the WALA project to SAFE's and create a *PointerAnalysis* object of the whole program using the *JSCallGraphBuilderUtil* class in WALA. Then by traversing over the *InstanceKeys* of the PointerAnlysis object we map the InstancesKeys' source locations to their *creation sites*. Similarly, by iterating over SAFE's control flow graph (CFG), we compute the mapping of source locations to SAFE's object *allocation sites*. Finally, we only consider the source locations present in both maps and take the InstanceKeys objects with their corresponding allocation site in SAFE.

As the set of the modeled built-in JavaScript functions in WALA and SAFE are not the same, we only consider user-defined program object locations. Additionally, the representation of local variables and lexical variables differs between WALA and SAFE, and does not provide similar numbers and types of pointers. Hence, only the user-defined global variables, user objects and argument objects are selected as appropriate candidates for our average points-to comparison. For instance, the global objects o_1, o_2 and x in Figure 3.1a have the same source locations in both analyzers and they are selected for the points-to comparison. The argument and function prototype objects are compared and selected in a similar fashion.

Average Points-to Set Computation in SAFE

To compute the average points-to set in SAFE, we take the heap at the *exit statement* of the top-level function. Since SAFE uses recency-abstraction for allocations in the heap, we compute an average effect of the old and recent objects of an allocation site to make it more comparable to WALA that does not rely on recency-abstraction. Afterwards, we iterate

over all user and global allocation site objects, and compute the average points-to set of all properties. However, only user-defined properties are considered while iterating over the global object. Additionally, the *length* property and properties that points-to *undefined* are ignored in all objects. The length property indicates the number of arguments and is not relevant. The points-to set for properties that point to undefined is zero in WALA and can falsely increase the precision impact of other pointers that point to more than one object. Hence, we ignore properties that point to undefined in both analyzers. However, the number of undefined properties are separately computed to estimate the percentage of JavaScript constructs that are not modeled in SAFE and WALA.

Average Points-to Set Computation in WALA

In WALA, we compute the heap from the points-to set of the InstanceKeys and select the heap objects which have a corresponding object in SAFE. The source location is used for filtering. After identifying the relevant objects, we iterate over all properties of each object to compute the average points-to set. In addition to the properties of these objects, the user-defined global variables and the callee function pointer variables are also considered. Because, unlike SAFE, WALA does not include the callee property in the arguments object. The same as in SAFE, the pointers pointing to zero (undefined) are computed separately to extract the percentage of unmodeled JavaScript constructs (functions).

3.4.2 Taint analysis

In this section, we extend SAFE's taint analysis technique, which models the sources of untrusted user inputs with the abstract value *string*[5] prototype value (see Figure 3.2), propagates it during the analysis and finally checks whether a tainted value might be used at the sink. In SAFE's existing taint analysis, the arguments of the sink functions are checked whether they are tainted (*strTop*) or not. However, this does not work for objects that contain tainted values in their property hierarchy, rather only for primitive values. Yet, finding only primitive argument values passed to a sink is rare. We did not encounter such a situation when analyzing the hybrid app from our industry partner.

To illustrate our contribution, let us consider the code in Figure 3.3. SAFE identifies taint flow only in $sink1$ but not in $sink2$ and $sink3$. In our implementation, we recursively iterate over the properties of the sink's argument objects and search for tainted values (strTop). Accordingly, we can find the taint flow to sink2 and sink3.

Some of the controllers in the hybrid app use *JSON.stringfy* to change the input data into a non-executable JSON string. However, this function is not currently modeled in SAFE. To analyze applications containing this function we extended SAFE's model such that our taint analysis approach correctly indicates that tainted value is not flowing to $sink4$, i.e., it

[5]string represents the top string value (*strTop*) in the lattice, not a regular string value. Other lattices could in principle be embedded into SAFE's to enable disambiguating more taint sources.

```
#Document.prototype: {
    [[Class]]: "Document",
    [[Extensible]]: true,
    [[Prototype]]: #Node.prototype,
    "write": <#Document.prototype.write, F, T, T>,
    "URL": <string, F, T, T>
},

```

FIGURE 3.2: Modeling untrusted user input example

```
var url = strTop (tainted)
var input = {user: "user1", url: url}
var card = { header: headers, data: input}
sink1(url)
sink2(input)
sink3(card)
var sanitizedValue = JSON.stringfy(card)
sink4(sanitizedValue)

```

FIGURE 3.3: Tainted object flowing to sink

supports the sanitization of JSON objects using *JSON.stringfy*. The implementation of the taint analysis and the analyzer comparison is available on Github[6]

3.4.3 SimplePack

Browsers do not support CommonJS[7] module syntax. Hence, there have been many ongoing efforts to make the utilities available in npm accessible to the browser. Browserify is one of the most popular tools able to bundle CommonJS modules for the browser by concatenating the modules in a single file.

Although Browserify bundles CommonJs modules for browsers, the bundles are not well suited for static analysis. Moreover, none of the aforementioned static analysis frameworks supported programs written in CommonJS module syntax at the time that we started this research effort. Figure 3.4 depicts the Browserify bundle program for the example in Figure 2.2. Although the bundle produced by Browserify works for browsers, it is not well suited for static analysis. The main problems with the analysis of Browserify bundles are:

1. Flow-insensitive points-to analysis will determine that the inner function may invoke module functions in any order.

2. At least one level of call-string sensitivity is needed to distinguish between different required modules, as a require call will first invoke the inner function.

[6]https://github.com/ghiwet/safe/tree/ subdirectories taint, walaPointsTo, safePointsTo
[7]CommonJS is a project with the goal of specifying an ecosystem for JavaScript outside the browser

```
(function outer (modules, cache, entry) { function inner (name) {
  if (!cache[name]) {
    if (!modules[name]) throw "MODULE_NOT_FOUND";
      var m = cache[name] = {exports:{}};
      modules[name][0].call(m.exports, function(x){
        var id = modules[name][1][x];
        return inner(id ? id : x);
      },m,m.exports);
    }
    return cache[name].exports;
  }
  inner(entry);
  return inner;
})
({1:[function(require,module,exports){
  ... // code of A.js here
},{}],2:[function(require,module,exports){
  ... // code of main.js here
},{"./A.js":1}]},{},2);

```

FIGURE 3.4: Simplified Browserify bundle for the Node.js module example in Figure 2.2

3. The additional function calls by the inner function renders the call graph overly complex and thus also requires a more context-sensitive static analysis.

Hence, we introduce *simplePack*, which bundles CommonJS modules in a more suitable way for static analysis frameworks. In our approach we concatenate the module functions and transform the require calls into the module functions to outer function calls, i.e., there is no inner function. Figure 3.5 illustrates the *simplePack* bundle for the same example in Figure 2.2.

In our implementation, we first compute the set of module dependencies by walking the dependency graph. In Node.js, the package *module-deps* resolves dependencies using node's module lookup algorithm. The AST of the empty program and for each module dependency is computed using the package *esprima*, which computes a SpiderMonkey AST. Then a module is wrapped in a function declaration and its AST is traversed using the *estraverse* package. During traversal every *require* call that has a string literal as argument[8] is replaced by a function call to that module. The potentially modified AST is added to the program. In the end, a function call to the entry module is added to the program and source code is

[8]Dynamically computed arguments are currently not supported and were not required for our analyses.

```
1 function _mod_A(module) {
2   var exports = module.exports;
3   function A() {
4     this.foo = function (x) { return x === 0; };
5   }
6   module.exports = A;
7   return module.exports;
8 }
9 function _mod_main(module) {
10  var exports = module.exports;
11  var A = _mod_A({ exports: {} });
12  var a = new A();
13  a.foo(2) // => false
14  return module.exports;
15 }
16 _mod_main({ exports: {} });
17
```

FIGURE 3.5: Simplified simplePack bundle for the Node.js module example in Figure 2.2

generated for it using the *escodegen* package. Our implementation is available on GitHub[9].

Algorithm 1: Pseudo Code for simple Pack

input : an entry module *file*
output: a bundle for *file*

```
1  modules ← Compute set of module dependencies for file;
2  programast ← Compute AST of an empty program;
3  foreach mod ∈ modules do
4      ast ← Compute AST for module mod;
5      fun ← Wrap ast in a function declaration;
6      Traverse fun and replace every require call by a function call to that module;
7      Add fun to programast;
8  end
9  Add function call to the entry module of programast;
10 program ← Generate source code from programast;
```

3.5 Evaluation

In this section, we evaluate the precision, scalability and code coverage trade-off of the static analyzers WALA and SAFE by comparing their average points-to set size and time taken for the analysis. Additionally, we evaluate the precision of our extended SAFE's taint analysis and the precision and recall of our Node.js modules bundler tool called *simplePack*

[9] https://github.com/MaxSchlueter/bundler

3.5.1 Points-to Analysis Comparison:

This section presents the evaluation of WALA's and SAFE's static program analyzers answering three research questions. First, we describe the research questions, evaluation methods and subjects (benchmarks). Then, we discuss the results of the analyses.

Research Questions:

We present the research question as follows:

RQ1. Precision: For the object properties (pointers keys) of each subject, to how many object locations or values do they point to on average? An analyzer that results in a lower average points-to value is more precise (provided that both analyses are sound).

RQ2. Scalability: How much time does it take to analyze a subject and how many subjects are fully analyzed within a given timeout? An analyzer that finishes the analysis of a subject in less time is more scalable.

RQ3. Coverage: How many pointer keys whose value is not undefined[10] does each analyzer identify for each subject? An analyzer that results in a higher number provides better coverage (unless it is less precise) as the points-to set is undefined when its pointer refers to an API function or object that is not modeled in the analysis.

Evaluation Methodology and subjects:

To answer these research questions we performed experiments using the WALA and SAFE 2.0 analyzers. For evaluation subjects, we used a version of the *hybrid app* from our industry partner and two benchmark sets from different categories [74, 167]: *addon* (i.e., plugins for the Firefox browser) and *standard* (i.e., from SunSpider and V8 browser benchmark suites). Each category of our benchmarks contains seven subjects [74, 167]. For all experiments, WALA used a 1-CFA sound propagation-based (PB) analysis with correlation tracking. SAFE analyzer uses 20-CFA and (10,5)-LSA, 10-length and 5-depth (to distinguish nested loops) loop strings, with recency abstraction for the hybrid app and the addon benchmark category. For the standard benchmark category, SAFE used 20-CFA and 0-LSA because loops are very complex and the analysis does not terminate in the given timeout with loop-sensitivity. WALA does not handle loops in any particular fashion except for *for-in* loops, which do not appear in these benchmarks. In the version of the industrial partner's app we replaced libraries like Lodash and jQuery by equivalent JavaScript code, and Promises by regular callbacks. We also removed certain code that is independent of the core functionality. Also, there is no model for platform-specific APIs that are written in a language other than JavaScript. This allows fair comparison but may miss certain security issues. These restrictions can be lifted by extending the analysis in future work.

[10] In WALA undefined is called zero.

TABLE 3.2: Precision and scalability comparison of analyzers. SLOC denotes the subjects' line of code without comments. The value under precision indicates the average points-to set whereas the values under the scalability are time of analysis in seconds. Entries marked χ denote that the analyzers do not finish analysis within the timeout of 10 minutes.

Subject (SLOC)		Precision		Scalability	
		SAFE	WALA	SAFE	WALA
hybrid-app (1221)		1.005	1.48	117	2.35
addon	odesk-job-watcher (154)	1.012	2.04	4.25	1.76
	chess (222)	1.02	1.34	8.4	2.1
	coffee-pods-deals (358)	1.006	1.46	2.5	2.3
	pinpoints (537)	1.02	1.44	3.55	2.58
	tryagain (590)	1.004	1.29	5.85	1.56
	less-spam-please (745)	1.028	1.29	11.2	2.6
	live-pagerank (865)	1.03	1.29	62	3.2
standard	access-nbody (170)	1.0	1.03	12.15	1.7
	crypto-sha1 (177)	1.0	1.085	24.5	1.8
	splay (201)	1.17	1.13	133	7.20
	richards (285)	1.48	5.79	206	2.8
	3d-cube (343)	1.11	3.97	380	4.88
	3d-raytrace (408)	χ	1.31	χ	1.23
	cryptobench (1297)	χ	χ	χ	χ

We conducted the experiments on a MacBook Pro with 2.9 GHz Intel Core i7 processor and 16GB RAM

Result and Discussion:

In this section, we discuss the precision, scalability and coverage result of the two analyzers. Table 3.2 depicts the precision and scalability evaluation result of the *hybrid app, addon and standard* subject categories.

Precision: To compare the precision of the two analyzers, we measure the average points-to set size of object properties over all user-defined locations (objects) and global variables. Although we consider only user-defined objects and global variables, we expect the results to be representative for the whole program. For all subjects that finish execution within the timeout, SAFE provides lower average points-to set size than WALA. For the *odesk-job-watcher*, *richards* and *3d-cube* subjects, WALA's average points-to set size exceeds SAFE's by a factor of two. Considering the average points-to set size over the benchmark categories, SAFE computes 1.017 for the addon and 1.15 for the standard category. In contrast, WALA determines 1.45 for addon and 2.6 for standard. Last but not least, the average points-to set size over all subjects is 1.073 for SAFE and 1.93 for WALA. This demonstrates that SAFE

TABLE 3.3: Coverage and % of undefined pointers comparison of analyzers. The % of undefined measures the percentage of pointers pointing to unmodeled API functions or objects, SLOC denotes the subjects' line of code without comments, and the entries marked χ denote that the analyzers do not finish analysis within the timeout of 10 minutes.

Subject (SLOC)		Coverage		% of undefined	
		SAFE	WALA	SAFE	WALA
hybrid-app (1221)		2391	2835	0%	0%
addon	odesk-job-watcher (154)	159	112	0%	12.5%
	chess (222)	265	135	0.4%	1.2%
	coffee-pods-deals (358)	313	287	0.6%	1.7%
	pinpoints (537)	440	339	0.9%	3%
	tryagain (590)	503	555	1.4%	.54%
	less-spam-please (745)	725	623	0.8%	11.4%
	live-pagerank (865)	818	702	0.6%	4.2%
standard	access-nbody (170)	195	150	11.51%	3.23%
	crypto-sha1 (177)	226	59	0.04%	7.8%
	splay (201)	156	20601	0%	0%
	richards (285)	339	491	0%	0%
	3d-cube (343)	1865	3252	0%	0%
	3d-raytrace (408)	χ	564	χ	2.5%
	cryptobench (1297)	χ	χ	χ	χ

provides more precise analysis than WALA. The improved precision of SAFE is due to its support for flow-sensitivity, context-sensitivity and loop-sensitivity in its analyses

Scalability: We evaluated the scalability of both analyzers by measuring the analysis time in seconds. For all subjects, WALA requires less time to finish analysis, indicating that it is more scalable. Analysis of *3d-raytrace* in SAFE did not terminate within the given timeout although it did in WALA. The scalability of the WALA analysis is due to its flow-insensitive points-to analysis approach. However, there is a cost in terms of precision which may be more relevant for security analysis.

Coverage: To compare the coverage of the analyzers, we counted the number of object properties for each subject. Properties pointing to non-undefined values, and those pointing to undefined are counted separately. Table 3.3 shows the coverage evaluation representing the pointer keys pointing to undefined in percentage with respect to the total pointer keys. For the majority of the subjects, SAFE computes higher non-undefined and lower undefined pointers, indicating that it models more API functions. The model coverage and the loop sensitivity has enormous effect on the number of pointers. In some cases, the number of non-undefined pointers is higher in WALA due to heap cloning, which creates different objects based on the context, for the same object locations. However, this happened only for

around 30 % of the subjects. Therefore, this indicates that in general SAFE provides better coverage.

Threats to Validity: The following are identified as threats to validity.

- The subjects used for our analysis may not be representative for other JavaScript applications.
- The analysis is based only on the user program and does not consider local variables (as these are specific to the analyzer's intermediate representation), which might deviate the average points-to set size over the whole application.

To generalize the comparison of both analyzers, we have listed some of the comparison features in Table 3.4.

3.5.2 Taint Analysis

In this section, we evaluate our taint analysis by investigating whether it identifies objects containing tainted values and considers sanitization using JSON.stringify. This evaluation aims to know how relevant our extended SAFE's taint analysis is in detecting potential injection vulnerabilities.

For the evaluation, we use the hybrid app. The hybrid app contains different controllers for login (configuration setup and automatic login using saved credentials) , *signin* (login by entering user credentials), *register* and *assistance*. All these controllers take user input and interact with the internet using HTTPS requests and post APIs. Hence, we took the program slice of each controller (i.e. all the code that can influence its computation) bundling all modules related to the respective controller's semantics, creating four separate programs to evaluate our taint analysis on. The user inputs at the respective controllers' form fields are considered a source of tainted value and the HTTPS request and post APIs are considered to be sinks. We use the abstract value *@StrTop* to simulate a tainted value.

Table 3.5 illustrates the result of the four program slices of the hybrid app, which all have more than 4,000 lines of code (without comments, SLOC). The result indicates that the

TABLE 3.4: Feature comparison of SAFE and WALA

Features	**SAFE**	**WALA**
Flow-sensitivity	Flow-sensitive	Flow-insensitive
Context-sensitivity	Any number	0 or 1-CFA
Loop-sensitivity	Any number	Only for-in
Precision	Relatively precise	Imprecise
Code Complexity	Easier to understand	Complex to understand
Soundiness [90]	Sound	Sound(PB) and unsound(FB)
Scalability	Less scalable	More scalable

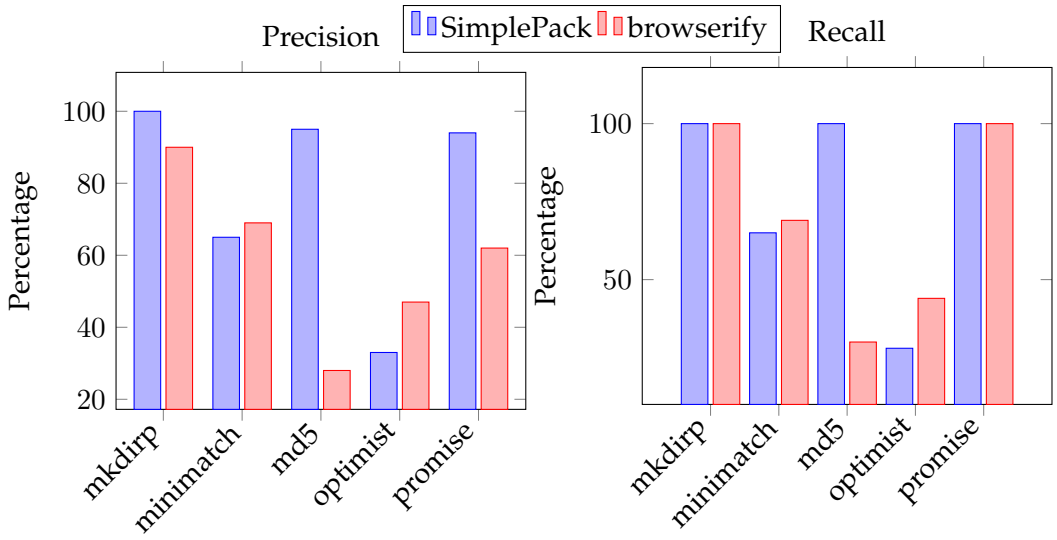


FIGURE 3.6: Precision and recall in percent for simplePack and Browserify bundles

tainted value is identified at the sink in both the login and signin program slices. Note that the application is still safe due to the way the tainted input is sent to the database in the platform code, which we verified by manual inspection. For the register and assistance program slices, no tainted value is identified at the sink. Upon manual investigation, we found that *JSON.stringify* is used to endorse the input before reaching the sink (i.e., we modeled *JSON.stringify* in SAFE in a way that can also simulate a sanitizer). When we remove this call to *JSON.stringify*, our taint analysis approach identifies the tainted input at the sink in both cases. Our manual investigation shows that the depth of the tainted properties in the parameter object is one or higher in all slices, which means that it could not be identified using SAFE's existing taint analysis. However, our extended taint analysis identifies the tainted objects.

3.5.3 SimplePack

To evaluate *simplePack*, we compare the static call graph (CG) of a bundled program to the dynamic one. This is done for programs transformed by simplePack and by Browserify. The

TABLE 3.5: Taint analysis result of four program slices from the real-world hybrid app.

program (SLOC)	Tainted	Sanitized	min−depth
login (4788)	yes	no	1
signin (4890)	yes	no	1
register (4501)	no	yes	2
assistance (4730)	no	yes	2

dynamic CG is constructed by a Jalangi2[11][131] analysis records all the functions that are invoked at a call site in one execution of the program. A further restriction in this evaluation is that only user-defined functions in the code are considered as call site targets. The static CG is constructed in WALA and conservatively approximates all the functions that can be invoked at some call site. The most precise static CG is the union of the dynamic CGs over all (potentially infinitely many) possible program executions.

The static and dynamic CGs are compared by computing the average precision and recall of all the call sites covered by the dynamic CG as in [48]. For a given call site the precision is the percentage of "true" function targets and recall is the percentage of correctly identified true targets with respect to all targets. For the evaluation we use five Node.js programs based on five different packages obtained from npm considering the list of most-depended packages. The selected packages have at least one module dependency, can be transformed by Browserify and simplePack and the resulting bundles are executed in node v0.12.x. Packages with dependency cycles or dynamic requires are excluded[12].

The average precision and recall of the bundles produced by simplePack and Browserify are depicted in Figure 3.6. Precision and recall are very high for the simplePack bundles of mkdirp, md5 and promise, achieving 100% recall and over 90% precision. The Browserify bundles for mkdirp and promise also achieve 100% recall but have lower precision measures than in simplePack. Recall and precision are particularly low for the md5 Browserify bundle when compared to the one by simplePack. The precision and recall measures for the mkdirp and optimist bundles are quite low and the Browserify bundles seem to be far better than their corresponding simplePack bundles. A recall measure less than 100% means that some call sites or function targets are missing in the static CG computed by WALA. Thus, the static analysis done by WALA for these bundles is unsound. Recall is at 65% for the minimatch simplePack bundle and at 69% for the corresponding Browserify bundle. Upon manual inspection we found that WALA was unable to resolve the same call sites and function targets in the Bundler bundle as in the corresponding Browserify bundle. Three unresolved call sites and their subsequent function calls lead to a low recall measure.

[11] https://github.com/Samsung/jalangi2

[12] This only happened for one package encountered in the list of most-depended packages and never for our industry application.

Chapter 4

Revealing Injection Vulnerabilities by Leveraging Existing Tests

4.1 Overview

In the high-profile 2017 *Equifax* attack, millions of individuals' private data was stolen, costing the firm nearly one and a half *billion* dollars in remediation efforts [130]. This attack leveraged a *code injection* exploit in Apache Struts (CVE-2017-5638) and is just one of over 9,711 similar code injection exploits discovered in recent years in popular software [100]. Code injection vulnerabilities have been exploited in repeated attacks on US election systems [143, 51, 95, 24], in the theft of sensitive financial data [132], and in the theft of millions of credit card numbers [78]. In the past several years, code injection attacks have persistently ranked at the top of the Open Web Application Security Project (OWASP) top ten most dangerous web flaws [105]. Injection attacks can be damaging even for applications that are *not* traditionally considered critical targets, such as personal websites, because attackers can use them as footholds to launch more complicated attacks.

In a code injection attack, an adversary crafts a malicious input that gets interpreted by the application as code rather than data. These weaknesses, "injection flaws", are so difficult to detect that rather than suggesting testing as a defense, OWASP suggests that developers try to avoid using APIs that might be targeted by attackers altogether or enforce site-wide input filtering. Consider again the Equifax hack: the underlying weakness that was exploited was originally introduced in 2011 and sat undetected in production around the world (not just at Equifax) for *six years* [99, 10]. While some experts blame Equifax for the successful attack — a patch had been released two months prior to the attack, but was not applied — one really has to ask: how is it possible that critical vulnerabilities go unnoticed in production software for so long?

With the exception of safety-critical and similar "high-assurance" software, general best practices call for developers to extensively test their applications, to perform code reviews,

and perhaps to run static analyzers to detect potentially weak parts of their software. Unfortunately, testing is a never-ending process: how do developers know that they've truly tested all input scenarios? To catch code injection exploits just-in-time, researchers have proposed deploying dynamic *taint tracking* frameworks, which track information flows, ensuring that untrusted inputs do not flow into sensitive parts of applications, *e.g.*, interpreters [138, 55, 147, 127, 17, 93]. However, these approaches have prohibitive runtime overheads: even the most performant can impose a slowdown of at least 10–20% and often far more [32, 17, 76, 45]. Although black-box fuzzers can be connected with taint tracking to detect vulnerabilities in the lab, it is difficult to use these approaches on stateful applications or those that require structured inputs [77, 58]. While some static analysis tools have seen recent developer adoption [52, 101, 18], statically detecting code injection vulnerabilities is challenging since static analysis tools must perform interprocedural data flow analysis [139, 158, 141, 13].

Our key idea is to use dynamic taint tracking *before deployment* to amplify developer-written tests to check for injection vulnerabilities. These integration tests typically perform functional checks. Our approach re-executes these existing test cases, *mutating* values that are controlled by users (*e.g.*, parts of each of the test's HTTP requests) and detecting when these mutated values result in real attacks. To our knowledge, this is the *first* approach that combines dynamic analysis with existing tests to detect injection attacks.

Key to our test amplification approach is a white-box *context-sensitive* input generation strategy. For each user-controlled value, state-of-the-art testing tools generate hundreds of attack strings to test the application [109, 108, 77]. By leveraging the context of *how* that user-controlled value is used in security-sensitive parts of the application, we can trivially rule out most of the candidate attack strings for any given value, reducing the number of values to check by orders of magnitude. Our testing-based approach borrows ideas from both fuzzing and regression testing, and is language agnostic.

We implemented this approach in the JVM, creating a tool that we call RIVULET. RIVULET Reveals Injection VUlnerabilities by Leveraging Existing Tests, and does not require access to application source code, and runs in commodity, off-the-shelf JVMs, integrating directly with the popular build automation platform *Maven*.

Like any testing-based approach, RIVULET is not guaranteed to detect all vulnerabilities. However, RIVULET guarantees that every vulnerability that it reports meets strict criteria for demonstrating an attack. We found that RIVULET performed as well as or better than a state-of-the-art static vulnerability detection tool [139] on several benchmarks. RIVULET discovers the Apache Struts vulnerability exploited in 2017 Equifax hack within minutes. When we ran RIVULET with the open-source project *Jenkins*, RIVULET found a previously unknown cross-site scripting vulnerability, which was confirmed by the developers. On

the educational project *iTrust* [56], RIVULET found 5 previously unknown vulnerabilities. Unlike the state-of-the-art static analysis tool that we used, Julia [139], RIVULET did not show *any* false positives.

Using dynamic analysis to detect injection vulnerabilities before deployment is hard, and we have identified two key challenges that have limited past attempts: (1) Unlike static analysis, dynamic analysis requires a representative workload to execute the application under analysis; and (2) For each potential attack vector, there may be hundreds of input strings that should be checked.

4.2 Contributions

RIVULET addresses the challenges mentioned above, making the following key contributions:

- A technique for re-using functional test cases to detect security vulnerabilities by modifying their inputs and oracles
- Context-sensitive mutational input generators for SQL, OGNL, and XSS that handle complex, stateful applications
- Embedded attack detectors to verify whether rerunning a test with new inputs leads to valid attacks

RIVULET is publicly available under the MIT license [62, 61].

4.3 Motivating Example

Injection vulnerabilities , such as SQL injection and cross-site scripting (XSS), enable attackers to insert different kinds of code into the target applications. Developers defend their software from such injection attacks through input validation and sanitization. Broadly, validation is a set of whitelisting techniques, such as: "only accept inputs that match a limited set of characters," while sanitization is a set of transformations that render attacks harmless, such as: "escape all quotation marks in user input." Ideally, each user-controlled

```
1@Override
2public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
3   String name = request.getParameter("name");
4   response.setContentType("text/html");
5   String escaped = StringEscapeUtils.escapeHtml4(name);
6   String content = "<a href=\"%s\">hello</a>";
7   try(PrintWriter pw = response.getWriter()) {
8     pw.println("<html><body>");
9     pw.println(String.format(content, escaped));
10    pw.println(String.format(content, name));
11    pw.println("</body></html>");
12  }
13}
```

LISTING 4.1: Two example XSS vulnerabilities. An untrusted user input from an HTTP request flows into the response to the browser on lines 9 and 10.

input (also referred to as a "tainted source") that can reach critical methods that may result in code execution (also referred to as a "sensitive sink") will be properly sanitized, validated, or both. Reaching such an ideal state is non-trivial [169, 89]. Hence, the key challenge in detecting these vulnerabilities is to detect flows from tainted sources to sensitive sinks that have not been properly sanitized.

Listing 4.1 shows a simplified example of two genuine cross-site scripting (XSS) vulnerabilities. XSS vulnerabilities allow an attacker to inject client-side scripting code into the output of an application which is then sent to another user's web browser. Lines 9 and 10 show a parameter provided by the user flowing into the response sent back to the browser *without* proper sanitization. In the first case (line 9), the vulnerability occurs despite an attempt to sanitize the user's input (using the Apache Commons-Language library function `escapeHtml4`), and in the second case (line 10), there is no sanitization at all.

In either case, providing the input string `javascript:alert('XSS');` for the parameter `"name"` will result in JavaScript code executing in the client's browser if they click on the link. The chosen sanitizer escapes any HTML characters in the input string (*i.e.,* preventing an injection of a `<script>` tag), but is insufficient for this case, as an attacker need only pass the prefix `javascript:` in their payload to cause code to execute when the user clicks on this link (many XSS attack payloads do not include brackets or quotes for this reason [109]).

To fix this vulnerability, the developer needs to apply a sanitizing function that prevents the insertion of JavaScript code. Static analysis tools, such as the state-of-the-art Julia platform [139], typically assume that library methods pre-defined as sanitizers for a class of attack (*e.g.,* XSS sanitizers) eliminate vulnerabilities for the data flows that they are applied to. In our testing-based approach, sanitizer methods do not need to be annotated by users. Instead we test whether a flow is adequately sanitized by attempting to generate a counterexample (*i.e.,* a malicious payload that produces a successful injection attack).

4.4 Methodology

Generating tests that expose the rich behavior of complicated, stateful web applications can be quite difficult. For instance, consider a vulnerability in a health records application that can only be discovered by logging in to a system, submitting some health data, and sending a message to a healthcare provider. Fuzzers have long struggled to generate inputs that follow a multi-step workflow like this example [77, 58]. Instead, RIVULET begins by executing the existing, ordinary test suite that developers have written, which does *not* need to have any security checks included in it: in this healthcare messaging example, an existing test might simply check that the workflow completes without an error. As we show in our evaluation (§5.4), even small test suites can be used by RIVULET to detect vulnerabilities.

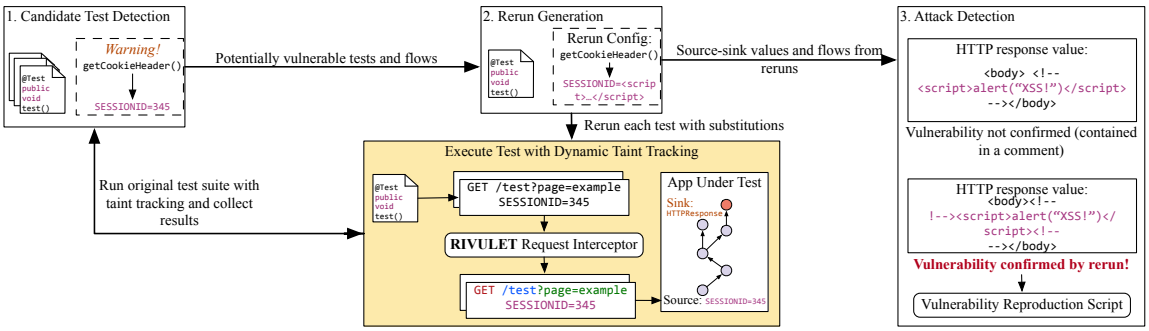


FIGURE 4.1: **High-Level Overview of RIVULET.** RIVULET detects vulnerabilities in three phases. Key to our approach is the repeated execution of developer-provided test cases with dynamic taint tracking. First, each developer-provided test is executed using taint tracking to detect which tests expose potentially vulnerable data flows. HTTP requests made during a test are intercepted and parsed into their syntactic elements which are then tainted with identifying information. Then, source-sink flows observed during test execution are recorded and passed with contextual information to a rerun generator. The rerun generator creates rerun configurations using the supplied flow and contextual information, and executes these reruns, swapping out developer-provided inputs for malicious payloads. Source-sink flows observed during test re-execution are passed to an attack detector which verifies source-sink flows that demonstrate genuine vulnerabilities.

Figure 4.1 shows a high-level overview of RIVULET's three-step process to detect injection vulnerabilities in web applications. First, RIVULET uses dynamic taint tracking while running each test to observe data flows from "sources," untrusted system inputs controlled by a potentially malicious actor, to "sinks," sensitive parts of an application that may be vulnerable to injection attacks. These source-sink flows do not necessarily represent vulnerabilities: it is possible that a sanitizer function correctly protects the application. Hence, when a source-sink flow is observed, RIVULET generates malicious payloads based on contextual information of the sink method. Then, tests are re-executed and those untrusted source values are replaced with generated payloads, probing for weak or missing sanitizers. Lastly, specialized logic based on the type of vulnerability, *e.g.*, XSS, is used as an oracle to determine whether a re-execution demonstrates a successful attack, thereby transforming a functional test into a security test.

In this way, source-sink flows are verified as vulnerable only if a successful attack can be demonstrated using a concrete exploit. This standard produces few false positives. Test reruns enable our technique to consider input sanitization and validation without requiring sanitization and validation methods to be explicitly specified or modeled. Verifying whether a sanitizer or validator is correct in all cases is a hard problem and beyond the scope of this work. However, if a system sanitizes or validates input improperly before it flows into a sink method, then one of the malicious payloads may be able to demonstrate a successful

attack, causing the flow to be verified. Our implementation of RIVULET (described in §4.5) automatically detects SQL injection, remote code execution, and cross-site scripting vulnerabilities: developers do not need to specify any additional sources or sinks in order to find these kinds of vulnerabilities. Section 4.5 describes, in detail, the specific strategy that RIVULET uses to find these kinds of vulnerabilities.

4.4.1 Detecting Candidate Tests

RIVULET co-opts existing, functional test cases to test for security properties by mutating user-controlled inputs and adding security-based oracles to detect code that is vulnerable to injection attacks. We assume that developers write tests that demonstrate typical application behavior, and our approach relies on automated testing to detect weak or missing sanitization. This assumption is grounded in best practices for software development: we assume that developers will implement some form of automated functional testing before scrutinizing their application for security vulnerabilities. RIVULET detects candidate tests by executing each test using dynamic taint tracking, identifying tests that expose potentially vulnerable source-sink flows, each of which we refer to as a *violation*. By leveraging developer tests, our approach can detect vulnerabilities that can only be revealed through a complex sequence of actions. These vulnerabilities can be difficult for test generation approaches to detect, but are critical when dealing with stateful applications [77].

In this model, developers do *not* need to write test cases that demonstrate an attack — instead, they need only write test cases that expose an information flow that is vulnerable to an attack. For instance, consider a recent Apache Struts vulnerability (CVE-2017-9791) that allowed user-provided web form input to flow directly into the Object-Graph Navigation Language (OGNL) engine. Struts includes a sample application for keeping track of the names of different people, this application can be used to demonstrate this vulnerability by placing an attack string in the "save person" form. To detect this vulnerability, we do *not* need to observe a test case that uses an attack string in the input, instead, we need only observe *any* test that saves *any* string through this form in order to observe the insecure information flow. Once this is detected, RIVULET, can then re-execute and perturb the test case, mutating the value of the form field, eventually demonstrating the exploit.

4.4.2 Rerun Generation and Execution

The next phase in RIVULET's vulnerability detection process is to re-execute each test, perturbing the inputs that the server received from the test case in order to add malicious payloads. A significant challenge to our approach is in the potentially enormous number of reruns that RIVULET needs to perform in order to test each potentially vulnerable source-sink flow. If an application has *thousands* of tests, each of which may have *dozens* of potentially vulnerable flows, it is crucial to limit the number of times that each test needs to be perturbed and re-executed. Unfortunately, it is typical to consider over *100* different malicious XSS payloads for *each* potentially vulnerable input [109, 77], and other attacks

may still call for dozens of malicious payloads.

Instead, RIVULET uses a white-box, in situ approach to payload generation in order to drastically reduce the number of reruns needed to evaluate a source-sink flow. Successful injection attacks often need to modify the syntactic structure of a query, document or command from what was intended by the developer [146]. By looking at the placement of taint tags (representing each source) within structured values that reach sink methods, *i.e.*, the syntactic context into which untrusted values flow, the number of payloads needed to test a flow can be limited to only those capable of disrupting that structure from the tainted portions of the value.

For instance, when an untrusted value reaches a sink method vulnerable to SQL injection attacks, developers usually intend for the value to be treated as a string or numeric literal. Consider the following SQL query: `SELECT * FROM animals WHERE name = '%Tiger%';` where the word `Tiger` is found to be tainted. In order to modify the structure of the query, a payload must be able to end the single-quoted string literal containing the tainted portion of the query. Payloads which do not contain a single-quote would be ineffective in this context, *e.g.*, payloads that aim to end double-quoted string literals, and do not need to be tested when evaluating this flow. RIVULET uses a similar approach for generating payloads for other kinds of attacks, as we will describe in § 4.5.2.

4.4.3 Attack Detection

The attack detector component provides the oracle for each modified test (removing any existing assertions), determining if the new input resulted in a successful attack on the system under test. There is a natural interdependence between payload generation and attack detection. Attack detection logic must be able to determine the success of an attack using any of the payloads that could be generated by RIVULET. Likewise, generated payloads should aim to trigger a successful determination from the detection logic. This relationship can be used not only to guide payload generation, but also to enable stricter (and simpler to implement) criteria for determining what constitutes a successful attack. Specifically, it is not necessary to recognize all possible successful attacks, but instead, only those generated by the system. Furthermore, this reduces the difficulty of formulating an appropriate detection procedure, particularly for certain types of attacks. RIVULET's attack detectors inspect both the taint tags and concrete values of data that reaches sensitive sinks.

4.5 Implementation

Our implementation of RIVULET for Java is built using the Phosphor taint tracking framework [17], and automatically configures the popular build and test management platform *Maven* to perform dynamic taint tracking during the execution of developer-written tests, generate malicious payloads based on source-sink flows observed during test execution, and execute test reruns. Developers can use RIVULET by simply adding a single maven

extension to their build configuration file: RIVULET and Maven automatically configure the rest. Out of the box, RIVULET detects cross-site scripting, SQL injection, and OGNL injection vulnerabilities without any additional configuration. Phosphor propagates taint tags by rewriting Java bytecode using the ASM bytecode instrumentation and analysis framework [110], and does not require access to application or library source code. We chose Phosphor since it is capable of performing taint tracking on all Java data types, ensuring that RIVULET is not limited in its selection of source and sink methods to only methods that operate on strings.

4.5.1 Executing Tests with Dynamic Tainting

RIVULET's approach for dynamic taint tracking within test cases is key to its success. Taint tracking allows data to be annotated with labels (or "taint tags"), which are propagated through data flows as the application runs. It is particularly critical to determine *where* these tags are applied (the "source methods") and how they correspond to the actual input that could come from a user, since it is at these same source methods that RIVULET injects malicious values when rerunning tests.

Many approaches to applying taint tracking to HTTP requests in the JVM use high-level Java API methods as taint sources, such as `ServletRequest.getParameter()` for parameters or, for cookies, `HttpServletRequest.getCookies()` [139, 96, 33, 54]. However, these approaches can be brittle: if a single source is missed or a new version of the application engine is used (which adds new sources), there may be false negatives. Moreover, since application middleware (between the user's socket request and these methods) performs parsing and validation, mutating these values directly could result in false positives when replaying and mutating requests. If RIVULET performed its injection *after* the middleware parses the HTTP request from the socket (*i.e.,* as a user application reads a value from the server middleware), RIVULET might generate something that could never have passed the middleware's validation. For instance, if performing a replacement on the method `getCookies()`, RIVULET might try to generate a replacement value `NAME=alert(String.fromCharCode(88,88,83))`, which could *never* be a valid return value from this method source, since HTTP cookies may not contain commas [97].

Instead of using existing Java methods as taint sources, RIVULET uses bytecode instrumentation to intercept the bytes of HTTP requests directly as they are read from sockets. Intercepted bytes are then buffered until a full request is read from the socket. Requests read from the socket are parsed into their syntactic elements, *e.g.,* query string, entity-body, and headers. Each element then passes through a taint source method which taints the characters of the element with the name of the source method, the index of the character in the element, and a number assigned to the request that was parsed. The original request is then reconstructed from the tainted elements and broken down back into bytes which are

passed to the object that originally read from the socket. This technique allows a tainted value to be traced back to a range of indices in a syntactic element of a specific request. Thus, this tainting approach enables precise replacements to be made during test re-executions.

We have integrated RIVULET with the two most popular Java HTTP servers, Tomcat [11] and Jetty [153], using bytecode manipulation. RIVULET modifies components in Tomcat and Jetty which make method calls to read bytes from a network socket to instead pass the receiver object (*i.e.*, the socket) and arguments of the call to the request interceptor. The interceptor reads bytes from any socket passed to it, parses the bytes into a request and taints the bytes based on their semantic location within the parsed request. It would be easy to add similar support to other Java web servers, however, Tomcat and Jetty are the most popular platforms by far.

4.5.2 Rerun Generation

RIVULET uses an easy-to-reconfigure, predefined set of sink methods, which we enumerate by vulnerability type below. When a sink method is called, the arguments passed to the call are recursively checked for taint tags, *i.e.*, arguments are checked, the fields of the arguments are checked, the fields of the fields of arguments checked, and so on until to a fixed maximum checking depth is reached. If a tainted value is found during the checking process, a source-sink flow is recorded. When RIVULET finishes checking the arguments of the call, it passes contextual information and flow information to a generator that handles the type of vulnerability associated with the sink method that was called. The contextual information consists of the receiver object of the sink method call and the arguments of the call. The flow information consists of the source information contained in the labels of the tainted values that were found and a description of the sink method that was called.

Rerun generators create rerun configurations identifying the test case that should be rerun, the detector that should be used to determine whether a successful attack was demonstrated by the rerun, the original source-sink flow that the rerun is trying to verify, and at least one replacement. Replacements define a replacement value, information used to identify the source value that should be replaced (target information), and possibly a ”strategy” for how the source value should be replaced. A replacement can either be built as a ”payload” replacement or a ”non-payload” replacement.

Payload replacements are automatically assigned target information and sometimes a strategy based on flow information. For example, the labels on a tainted value that reached some sink might show that the value came from indices $6-10$ of the second call to the source `getQueryString()`. One payload replacement built off of that flow information would direct that the second time `getQueryString()` is called that its return value should be replaced using a strategy that replaces only indices six through ten with a replacement value. Payload replacements are how malicious payloads are normally specified, thus

every rerun is required to have at least one of them. Non-payload replacements are useful for specifying secondary conditions that an attack may need in order to succeed, such as changing the "Content-Type" header of a request.

SQL Injection. The rerun generator for SQL injection uses all `java.sql.Statement` and `java.sql.Connection` methods that accept SQL code as sinks, and considers three primary SQL query contexts in which a tainted value may appear: literals, comments, LIKE clauses. Tainted values appearing in other parts of the query are treated similarly to unquoted literals. Tainted values appearing in LIKE clauses are also considered to be in literals, thus cause both the payloads for tainted literals and tainted LIKE clauses to be generated. If a tainted value appears in a literal, the generator first determines the "quoting" for the literal. A literal can be either unquoted (like a numeric literal might be), single-quoted, double-quoted, or backtick-quoted (used for table and column identifiers in MySQL). Payloads for tainted literals are prefixed by a string that is based on the quoting of the literal in order to attempt to end the literal. The quoting can also be used to determine an appropriate ending for payloads. If a tainted value appears in a comment, the generator first determines the characters used to end and start the type of comment the value appears in. Payloads for tainted comments are prefixed by the end characters for the comment and ended with the start characters for the comment. If a tainted value appears in a LIKE clause, the generator creates payloads containing SQL wildcard characters.

RIVULET generates 2–5 SQL injection payloads for a tainted value in a particular context out of 20 unique payloads that could be generated for the same tainted value across all of the contexts considered by the SQL injection rerun generator. If wildcard payloads for LIKE clause are not generated then only 2–3 payloads are generated per context. This is a reduction from Kieżun *et al.*'s *Ardilla*, which uses 6 SQL injection patterns and does not consider tainted backtick-quoted values, comments, or LIKE clauses [77].

Cross-Site Scripting. RIVULET uses special sink checking logic for XSS, checking data as it is sent over-the-wire to the browser. The overloaded variants of `SocketChannel.write()` are used as sink methods for XSS attacks. In order to give the XSS generator all of the HTML content for a single response at once, RIVULET stores the bytes written to a socket until a full response can be parsed from the bytes. If the parsed response contains HTML content and the HTML in the response's entity-body contains a tainted value, then that HTML is passed to the XSS rerun generator.

The XSS rerun generator parses HTML content into an HTML document model using the Jsoup library [71]. This model is traversed, generating payloads for each tainted value encountered. The XSS rerun generator considers 5 primary HTML document contexts in which a tainted value may appear: tag names, attribute names, attribute values, text or data content, and comments. Different payloads are capable of introducing a script-triggering

mechanism into the document's structure depending on the context. RIVULET also addresses context-specific issues like the quoting of attribute values or whether content is contained in an element which causes the tokenizer to leave the data state during parsing [173].

The XSS generator also considers whether a tainted value was placed in a context that would already be classified as an embedded script or the address of an external script. Furthermore, if a tainted value appears in a context that would be classified as an embedded script then the generator also determines whether the tainted value is contained within a string literal, template literal, or comment.

RIVULET generates 3–7 XSS payloads for a tainted value in a particular context out of over 100 unique payloads that could be generated for the same tainted value across all of the contexts considered by the XSS rerun generator. By comparison, OWASP's "XSS Filter Evasion Cheat Sheet" features 152 unique payloads for cross-site scripting attacks [109] and Ardilla uses 106 patterns for creating cross-site scripting attacks [77].

Command and OGNL Injection. The command injection rerun generator creates payloads with common UNIX commands like `ls`, considering `java.lang.ProcessBuilder` and `java.lang.Runtime` methods as sinks.

The OGNL injection rerun generator creates payloads that facilitate attack detection. It can be difficult to specify generic criteria for detecting any OGNL injection attack because the language is designed to allow users to execute "non-malicious" code. OGNL expressions can modify Java objects' properties, access Java objects' properties and make method calls [151]. Applications using OGNL can limit the code that user specified expressions can execute by whitelisting or blacklisting certain patterns [152]. The evaluation of improperly validated OGNL expressions can enable a user to execute arbitrary code. The OGNL rerun generator uses payloads that we collected from the Exploit Database [47] and simplified to integrate more tightly with RIVULET's attack detection mechanism.

Rerun Execution. Rerun configurations created by the rerun generators specify test cases that should be re-executed. Values are replaced when they are assigned a label at a source method and the information on the label being assigned to the value meets the criteria specified by one of the current rerun configuration's replacements. Replacements may dictate a strategy for replacing the original value; strategies can specify ways of combining an original value with a replacement value, a way of modifying the replacement value, or both. For example, a strategy could specify that only a certain range of indices in the original value should be replaced, that the replacement value should be percent encoded, or both. RIVULET automatically converts values to ensure that the type of a replacement value is appropriate (*e.g.*, converting between a string and a character array).

4.5.3 Attack Detection

Rerun configurations specify which vulnerability-specific attack detector should be used to check flows during a test re-execution.

SQL Injection. Our approach for detecting SQL injection attack builds on Halfond *et al.*'s "syntax-aware evaluation" model, which calls for checking that all parts of SQL queries except for string and numeric literals come from trusted sources [55]. We determine a SQL injection attack to be successful if a tainted SQL keyword not contained in a literal or comment is found within a query that reached a sink vulnerable to SQL injection. Alternatively, an attack is deemed successful if a sink-reaching query contains a `LIKE` clause with an unescaped tainted wildcard character (*i.e.*, % or _) as the system could be vulnerable to a SQL wildcard denial-of-service attack [108]. The attack detector for SQL injection uses ANTLR, a parser generation tool [150] and JSqlParser, a SQL statement parser that supports multiple SQL dialects [72], to parse SQL statements that reach sink methods vulnerable to SQL injection attacks.

Cross-Site Scripting. The World Wide Web Consortium's (W3C's) Recommendation for HTML 5.2 specifies mechanisms which can trigger the execution of embedded or external scripts: "processing of script elements," "navigating to javascript: URLs," "event handlers," "processing of technologies like SVG that have their own scripting features" [172]. Only the syntactic components of an HTML document that are capable of activating a script-triggering mechanism are vulnerable to script injections. As such, we determine the success of an XSS attack by checking these vulnerable components.

RIVULET intercepts and buffers the bytes of HTTP responses until a full response can be parsed from the bytes. Then, the parsed document is checked for components that could activate a script-triggering mechanism. Depending on the mechanism potentially activated by the component, a portion of the component is then classified as either an embedded script or the address of an external script. The following rules are used to identify embedded and external scripts in the response: (1) The inner content of every "script" tag is classified as an embedded script. (2) The HTML entity decoded value of every "src" attribute specified for a "script" tag is classified as an external script's address. (3) The HTML entity decoded value of every "href" attribute specified for a "base" tag is classified as an external script's address because of its potential impact on elements in the document using relative URLs. (4) The HTML entity decoded value of every event handler attribute, *e.g.*, "onload," specified for any tag is classified as an embedded script. (5) The HTML entity decoded value of every attribute listed as having a URL value in W3C's Recommendation for HTML 5.2 [172], *e.g.*, the "href" attribute, is examined. If the decoded value starts with "javascript:", then the portion of the decoded value after "javascript:" is classified as an embedded script.

Embedded scripts are checked for values successfully injected into non-literal, non-commented portions of the script. To do so, the portions of the script that are not contained in JavaScript string literals, template literals, or comments are checked for a predefined target string. This target string is based on the malicious payload being used in the current test re-execution, *e.g.*, `alert` is an appropriate target string for the payload `<script>alert(1)</script>`, but other payloads may have more complicated target strings. If the target string is found in the non-literal, non-commented portions of the script and it is tainted, then the attack is deemed successful. Since the target string must be tainted to be deemed a successful attack, a vulnerability will be reported only if an attacker could inject that target string into the application.

External script addresses are checked for successfully injected URLs that could potentially be controlled by a malicious actor. The start of each address is checked for a predefined target URL. The target URL is based on the malicious payload being used in the current test re-execution. If the target URL is found at the start of an address and is tainted, then the attack is deemed successful.

The XSS attack detector stores bytes written to a socket by calls to `SocketChannel.write()` until a full response can be parsed (using Jsoup [71]) from the bytes stored for a particular socket. The rules described above are then applied to the document model parsed from the entity-body. The embedded script checks are also performed using ANTLR [150] and a simplified grammar for JavaScript to identify string literals, template literals, and comments.

Command and OGNL Injection. A command injection attack is determined to be successful if any tainted value flows into a sink vulnerable to command injection (such as `ProcessBuilder.command()` and `Runtime.exec()`). Additionally, if a call is made to `ProcessBuilder.start()`, the detector will deem an attack successful if the "command" field of the receiver object for the call is tainted. This relatively relaxed standard is a product of a lack of legitimate reasons for allowing untrusted data to flow into these sinks and the severity of the security risk that doing so presents. This approach could be fine-tuned to perform more complicated argument parsing (similar to the XSS detector), however, in practice, we found it sufficient, producing no false positives on our evaluation benchmarks. We use a similar tactic to test for successful OGNL injection attacks since the OGNL payloads generated by RIVULET are crafted to perform command injection attacks.

4.5.4 Limitations

Our approach is not intended to be complete; it is only capable of detecting vulnerabilities from source-sink flows that are exposed by a test case. Hence, RIVULET requires applications to have existing test cases, although we believe that this is a fair assumption to make, and in our evaluation, show that RIVULET can detect a real vulnerability even when presented with

TABLE 4.1: Comparison of RIVULET and Julia [139] on third-party benchmarks. For each vulnerability type in each benchmark suite we show the total number of test cases (for both true and false alarm tests). For RIVULET and Julia, we report the number of true positives, false positives, true negatives, false negatives, and analysis time in minutes. Times are aggregate for the whole benchmark suite.

		# Test Cases		RIVULET					Julia				
Suite	**Type**	**True Alarm**	**False Alarm**	**TP**	**FP**	**TN**	**FN**	**Time**	**TP**	**FP**	**TN**	**FN**	**Time**
Juliet	RCE	444	444	444	0	444	0	25	444	0	444	0	33
	SQL	2,220	2,220	2,220	0	2,220	0		2,220	0	2,220	0	
	XSS	1,332	1,332	1,332	0	1,332	0		1,332	0	1,332	0	
OWASP	RCE	126	125	126	0	125	0	3	126	20	105	0	15
	SQL	272	232	272	0	232	0		272	36	196	0	
	XSS	246	209	246	0	209	0		246	19	190	0	
Securibench-Micro	SQL	3	0	3	0	0	0	1	3	0	0	0	1
	XSS	86	21	85	0	21	1		77	14	7	9	
WavSep	SQL	132	10	132	0	10	0	2	132	0	10	0	2
	XSS	79	7	79	0	7	0		79	6	1	0	

a very small test suite (for Apache Struts). This limitation could be mitigated by integrating our approach with an automatic test generation technique. Vulnerabilities caused by a nondeterministic flow are hard for RIVULET to detect, even if the flow occurs during the original test run, because the flow may fail to occur during the re-execution of the test. RIVULET does not detect XSS attacks which rely on an open redirection vulnerability [154]. More generally, RIVULET can only detect attacks that we have constructed generators and detectors for, but this is primarily a limitation of RIVULET's implementation, and not its approach. We note that even static analysis tools can only claim soundness to the extent that their model holds in the code under analysis: in our empirical evaluation of a sound static-analysis tool, we found that the static analyzer missed several vulnerabilities (§4.6.1).

Since Phosphor is unable to track taint tags through code outside of the JVM, RIVULET is also unable to do so. As a result, RIVULET cannot detect persistent XSS vulnerabilities caused by a value stored in an external database, but it can detect one caused by a value stored in Java heap memory. We plan to propose extensions to Phosphor to overcome this limitation, building off of work demonstrating the feasibility of persisting taint tags in databases in the Android-based TaintDroid system [149]. At present, RIVULET can only detect vulnerabilities that result from explicit (data) flow, and not through implicit (control) flows, or side-channels such as timing [124], a limitation shared by most other tools, including Julia [139]. Experimental support for implicit flow tracking in Phosphor may lift this limitation in the future. Despite these limitations, we have found RIVULET to be effective at detecting injection vulnerabilities.

4.6 Evaluation

We performed an empirical evaluation of RIVULET, with the goal of answering several research questions:

RQ1: How does RIVULET perform in comparison to a state-of-the-art static analysis tool?

RQ2: Does RIVULET scale to large projects and their test suites?

RQ3: How significantly does RIVULET's contextual payload generation reduce the number of reruns needed?

To answer these questions, we applied both RIVULET and the state-of-the-art static analysis tool Julia [139] to several suites of vulnerability detection benchmarks. These curated benchmarks are intentionally seeded with vulnerabilities, allowing us to compare RIVULET and Julia in terms of both precision and recall. We were also able to use one of these benchmarks to compare RIVULET against six commercial vulnerability detection tools. These benchmarks allow us to evaluate the efficacy of RIVULET's attack generators and detectors, but since they are micro-benchmarks, they do not provide much insight into how RIVULET performs when applied to real, developer-provided test suites. To this end, we also applied RIVULET to three larger applications and their test suites.

We conducted all of our experiments on Amazon's EC2 infrastructure, using a single "c5d.4xlarge" instance with 16 3.0Ghz Intel Xeon 8000-series CPUs and 32 of RAM, running Ubuntu 16.04 "xenial" and OpenJDK 1.8.0_222. We evaluated Julia by using the JuliaCloud web portal, using the most recent version publicly available as of August 16, 2019. When available (for Juliet-SQLI, Juliet-XSS and all of OWASP), we re-use results reported by the Julia authors [139]. When we executed it ourselves, we confirmed our usage of Julia through personal communication with a representative of JuliaSoft, and greatly thank them for their assistance.

4.6.1 RQ1: Evaluating RIVULET on Benchmarks

In order to evaluate the precision and recall of RIVULET and Julia, we turn to third-party vulnerability detection benchmarks, specifically NIST's Juliet Benchmark version 1.3 [98], OWASP's Benchmark version 1.2 [107], Livshits' securibench-micro [88], and the Application Vulnerability Scanner Evaluation Project's WAVSEP version 1.5 [31]. Each of these benchmarks contains test cases with vulnerabilities that are representative of real vulnerabilities found in various applications. From these tests, we can collect the number of true positives and false negatives reported by each tool. The benchmarks also contain test cases with variants of those vulnerabilities that are *not* vulnerable, allowing us to also collect the number of false positives and true negatives reported by each tool.

Each benchmark consists of a series of web servlets (and in some cases, also non-servlet applications) that are tests well-suited for analysis by a static analyzer like Julia. However, RIVULET requires executable, JUnit-style test cases to perform its analysis. Each servlet is designed to be its own standalone application to analyze, and they are not stateful. Hence, for each benchmark, we generated JUnit test cases that requested each servlet over HTTP,

passing along some default, non-malicious parameters as needed. Where necessary, we modified benchmarks to resolve runtime errors, mostly related to invalid SQL syntax in the benchmark. We ignored several tests from securibench-micro that were not at all suitable to dynamic analysis (some had infinite loops, which would not result in a page being returned to the user), and otherwise included only tests for the vulnerabilities targeted by RIVULET (RCE, SQLI and XSS). Most of these benchmarks have only been analyzed by static tools, and not executed, and hence, such issues may not have been noticed by prior researchers.

Table 4.1 presents our findings from applying both RIVULET and Julia to these benchmarks. RIVULET had near perfect recall and precision, identifying every true alarm test case as a true positive but one, and every false alarm test case as a true negative. In three interesting Securibench-Micro test cases, the test case was non-deterministically vulnerable: with some random probability the test could be vulnerable or not. In two of these cases, RIVULET eventually detected the vulnerability after repeated trials (the vulnerability was exposed with a 50% probability and was revealed after just several repeated trials). However, in the case that we report a false negative (simplified and presented in Listing 4.2), the probability of any attack succeeding on the test was just $1/2^{32}$, and RIVULET could not detect the vulnerability within a reasonable time bound. We note that this particularly difficult case does not likely represent a significant security flaw, since just like RIVULET, an attacker can not control the probability that their attack would succeed. This test case likely represents the worst-case pathological application that RIVULET could encounter.

In comparison, Julia demonstrated both false positives and false negatives. Many of the false positives were due to Julia's lack of sensitivity for multiple elements in a collection, resulting in over-tainting all elements in a collection. We confirmed with JuliaSoft that the tool's false negatives were not bugs, and instead generally due to limitations in recovering exact dynamic targets of method calls when the receiver of a method call was retrieved from the heap, causing it to (incorrectly) assume a method call to not be a sink. Listing 4.3 shows an example of one such case, where Julia reports a vulnerability on Line 3 but not on Line 6 since it is unable to precisely determine the dynamic target of the second `println`. Unlike the very tricky non-deterministic case that RIVULET struggled to detect, we note that this form of data flow is not uncommon, and this limitation may significantly impact Julia's ability to detect XSS vulnerabilities in applications that pass the servlet's `PrintWriter` between various application methods.

```
void doGet(HttpServletRequest req, HttpServletResponse resp) {
  Random r = new Random();
  if (r.nextInt() == 3)
    resp.getWriter().println(req.getParameter("name"));
}
```

LISTING 4.2: Simplified code of the vulnerability RIVULET misses. `r.nextInt()` returns one of the 2^{32} integers randomly.

We also collected execution times to analyze each entire benchmark for both tools. For RIVULET, we report the total time needed to execute each benchmark (including any necessary setup, such as starting a MySQL server), and for Julia, we report the execution time from the cloud service. Despite its need to execute thousands of JUnit tests, RIVULET ran as fast or faster than Julia in all cases. Most of RIVULET's time on these benchmarks was spent on the false positive tests, which act as a "worst case scenario" for its execution time: if RIVULET can confirm a flow is vulnerable based on a single attack payload, then it need not try other re-run configurations for that flow. However, on the false positive cases, RIVULET must try every possible payload (in the case of XSS, this is up to 7, although it may also try different encoding strategies for each payload, depending on the source).

Unfortunately, it is not possible to report a direct comparison between RIVULET and any commercial tools (except for Julia) due to licensing restrictions. However, the OWASP benchmark is distributed with anonymized results from applying six proprietary tools (Checkmarx CxSAST, Coverity Code Advisor, HP Fortify, IBM AppScan, Parasoft Jtest, and Veracode SAST) to the benchmark, and we report these results in comparison to RIVULET. Table 4.2 presents these results (each commercial tool is anonymized), showing the true positive rate and false positive rate for each tool. RIVULET outperforms each of these commercial static analysis tools in both true positive and false positive detection rates.

4.6.2 RQ2: RIVULET on Large Applications

While the benchmarks evaluated in § 4.6.1 are useful for evaluating the potential to detect vulnerabilities, they are limited in that they are micro-benchmarks. They help us make general claims about how RIVULET might perform when applied to an arbitrary application. However, since each micro-benchmark is designed to be easily executed (and indeed, we automatically generated tests to execute them), it is not possible to judge how RIVULET performs when using existing, *developer-written*, test cases on real applications.

To provide more detailed results on how RIVULET performs on larger, real applications, we applied it to three different open-source Java web applications and their existing JUnit test suites. *iTrust* is an electronic health record system iteratively developed over 25 semesters by students at North Carolina State University [56, 67]. We evaluated iTrust version 1.23,

```
1private PrintWriter writer;
2void doGet(HttpServletRequest req, HttpServletResponse resp) {
3  resp.getWriter().println(req.getParameter("dummy"));
4  //XSS reported on line above
5  this.writer = resp.getWriter();
6  this.writer.println(req.getParameter("other"));
7  //No XSS reported on line above
8}
```

LISTING 4.3: Example of a false negative reported by Julia

TABLE 4.2: Comparison between RIVULET and different vulnerability detection tools on the OWASP benchmark. For each vulnerability type, we report the true positive rate and false positive rate for the tool. Each *SAST-0** tool is one of: Checkmarx CxSAST, Coverity Code Advisor, HP Fortify, IBM AppScan, Parasoft Jtest, and Veracode SAST.

	RCE		**SQL**		**XSS**	
Tool	TPR	FPR	TPR	FPR	TPR	FPR
SAST-01	35%	18%	37%	13%	34%	25%
SAST-02	67%	42%	94%	62%	67%	42%
SAST-03	59%	35%	82%	47%	49%	22%
SAST-04	72%	42%	83%	51%	66%	40%
SAST-05	62%	57%	77%	62%	41%	25%
SAST-06	100%	100%	100%	90%	85%	45%
RIVULET	100%	0%	100%	0%	100%	0%

TABLE 4.3: Results of executing RIVULET on open-source applications. For each application we show the number of lines of Java code (as measured by `cloc` [36]) the number of test methods, and the time it takes to run those tests with and without RIVULET. For each vulnerability type, we show the number of potentially vulnerable flows detected by RIVULET (**Flows**), the naive number of reruns that would be performed without RIVULET's contextual payload generators (**Reruns**$_n$), the actual number of reruns (**Reruns**), the number of reruns succeeding in exposing a vulnerability (**Crit**), and the number of unique vulnerabilities discovered (**Vuln**). There were no SQL-related flows.

			Time (Minutes)		**RCE**					**XSS**				
Application	**LOC**	**Tests**	**Baseline**	**RIVULET**	**Flows**	**Reruns**$_n$	**Reruns**	**Crit**	**Vuln**	**Flows**	**Reruns**$_n$	**Reruns**	**Crit**	**Vuln**
iTrust	80,002	1,253	6	239	0	0	0	0	0	124	117,778	5,424	289	5
Jenkins	185,852	9,330	85	1,140	0	0	0	0	0	534	294,489	13,562	9	1
Struts Rest-Showcase	152,582	15	0.3	5	53	2,609	2,609	4	1	9	6,254	228	0	0

the most recent version of iTrust1 — a newer "iTrust2" is under development, but has far less functionality than iTrust1 [56]. A prior version of iTrust was also used in the evaluation of Mohammadi *et al.*'s XSS attack testing tool, although the authors were unable to provide a detailed list of the vulnerabilities that they detected or the specific version of iTrust used [96]. We also assessed a recent revision, `8349cebb`, of *Jenkins*, a popular open-source continuous integration server [69], using its test suite. *Struts* is an open-source web application framework library which is used to build enterprise software [9]. Struts is distributed with sample applications that use the framework, as well as JUnit tests for those applications. We evaluated RIVULET with one such sample application (*rest-showcase*), using Struts version 2.3.20_1, which is known to have a serious RCE vulnerability.

Table 4.3 presents the results of this experiment, showing for each project the number of tests, and then for each injection category the number of vulnerable flows, reruns executed, reruns that succeeded in finding a vulnerability, and the number of unique vulnerabilities found. RIVULET reported no false positives. We briefly discuss the vulnerabilities that RIVULET detected in each application below.

In *iTrust*, RIVULET detected five pages with XSS vulnerabilities, where a user's submitted form values were reflected back in the page. While these values were in only five pages, each page had multiple form inputs that were vulnerable, and hence, RIVULET reported a total of 289 different rerun configurations that demonstrate these true vulnerabilities. There were no flows into SQL queries in iTrust: while iTrust uses a MySQL database, it exclusively accesses it through correct use of the `preparedStatement` API, which is designed to properly escape all parameters. We reported all five vulnerabilities to the *iTrust* developers and submitted a patch.

We also submitted iTrust to the Julia cloud platform for analysis, which produced 278 XSS injection warnings. We did not have adequate resources to confirm how many of these warnings are false positives, but did check to ensure that Julia included all of the XSS vulnerabilities that RIVULET reported. We describe one example that we closely investigated and found to be a false positive reported by Julia. The vulnerability consists of a page with a form that allows the user to filter a list of hospital rooms and their occupants by filtering on three criteria. After submitting the form, the criteria submitted by the user are echoed back on the page without passing through any standard sanitizer, hence Julia raises an alert. While RIVULET did not alert that there was a vulnerability on this page, it did observe the same potentially vulnerable data flow, and generated and executed rerun configurations to test it (not finding it to be vulnerable). We carefully inspected this code to confirm that RIVULET's assessment of these flows was correct, and found that the filter criteria would only be displayed on the page if there were any rooms that matched those criteria. The only circumstances that an exploit could succeed here would be if an administrator had defined

a hospital or ward named with a malicious string — in that case, that same malicious string could be used in the filter. While perhaps not a best practice, this does not represent a serious risk — an untrustworthy administrator could easily do even more nefarious actions than create the scenario to enable this exploit.

In *Jenkins*, RIVULET detected a single XSS vulnerability, but that vulnerability was exposed by multiple test cases, and hence, RIVULET created 9 distinct valid test rerun configurations that demonstrated the vulnerability. We contacted the developers of Jenkins who confirmed the vulnerability, assigned it the identifier CVE-2019-10406, and patched it. Jenkins does not use a database, and hence, had no SQL-related flows. We did not observe flows from user-controlled inputs to command execution APIs. Jenkins' slower performance was caused primarily by its test execution configuration, which calls for every single JUnit test class to execute in its own JVM, with its own Tomcat server running Jenkins. Hence, for each test, a web server must be started, and Jenkins must be deployed on that server. This process is greatly slowed by load-time dynamic bytecode instrumentation performed by RIVULET's underlying taint tracking engine (Phosphor), and could be reduced by hand-tuning Phosphor for this project.

In *Struts*, RIVULET detected a command injection vulnerability, CVE-2017-5638, the same used in the Equifax attack (this vulnerability was known to exist in this revision). Again, multiple tests exposed the vulnerability, and hence RIVULET generated multiple rerun configurations that demonstrate the vulnerabilities. In this revision of struts, a request with an invalid HTTP Content-Type header can trigger remote code execution, since that header flows into the OGNL expression evaluation engine (CVE-2017-5638), and RIVULET demonstrates this vulnerability by modifying headers to include OGNL attack payloads. The struts application doesn't use a database, and hence, had no SQL-related flows.

The runtime for RIVULET varied from 5 minutes to about 19 hours. It is not unusual for automated testing tools (*i.e.,* fuzzers) to run for a full day, or even several weeks [79], and hence, we believe that even in the case of Jenkins, RIVULET's performance is acceptable. Moreover, RIVULET's test reruns could occur in parallel, dramatically reducing the wall-clock time needed to execute it.

4.6.3 RQ3: Reduction in Reruns

This research question evaluates RIVULET's reduction in the number of reruns needed to test whether a given source-sink flow is vulnerable to an attack compared to a naive approach. To do so, we considered the number of payloads that a more naive attack generator such as Ardilla [77] or Navex [4] might create for each class of vulnerability, and then estimate the number of reruns needed. To estimate the number of payloads used for XSS testing, we referred to the OWASP XSS testing cheat sheet, which has 152 distinct payloads [109]. We assume that for RCE testing, the naive generator would generate the same 12 payloads

that RIVULET uses (RIVULET does not use context in these payloads). We assume that the naive generator will also consider multiple encoding schemes for each payload (as RIVULET does). Hence, to estimate the number of reruns created by this naive generator, we divide the number of reruns actually executed by the total number of payloads that RIVULET could create, and then multiply this by the number of payloads that the naive generator would create (*e.g.*, $Reruns/7 * 152$ for XSS).

Table 4.3 shows the number of reruns generated by this naive generator as Reruns_n. As expected, RIVULET generates far fewer reruns, particularly with its XSS generator, where it generated $22x$ fewer reruns for Jenkins than the naive generator would have. Furthermore, given that RIVULET took 19 hours to complete on Jenkins, prior approaches that do not use RIVULET's in situ rerun generation would be infeasible for the project. Hence, we conclude that RIVULET's context-sensitive payload generators are quite effective at reducing the number of inputs needed to test if a source-sink flow is vulnerable to attack.

4.6.4 Threats to Validity

Perhaps the greatest threat to the validity of our experimental results comes from our selection of evaluation subjects. Researchers and practitioners alike have long struggled to establish a reproducible benchmark for security vulnerabilities that is representative of real-world flaws to enable a fair comparison of different tools [79]. Thankfully, in the context of injection vulnerabilities, there are several well-regarded benchmarks. To further reduce the threat of benchmark selection, we used four such benchmarks (Juliet, OWASP, Securibench-Micro and WavSep). Nonetheless, it is possible that these benchmarks are not truly representative of real defects — perhaps we overfit to the benchmarks. However, we are further encouraged because these benchmarks include test cases that expose the known limitations of both RIVULET and Julia: for RIVULET, the benchmark suite contains vulnerabilities that are exposed only non-deterministically, and for Julia, the benchmark suite contains tests that are negatively impacted by the imprecision of the static analysis. To aid reproducibility of our results, we have made RIVULET (and scripts to run the benchmarks) available under the MIT open source license [61, 62].

To demonstrate RIVULET's ability to find vulnerabilities using developer-written tests, we were unable to find any appropriate benchmarks, and instead evaluate RIVULET on several open-source projects. It is possible that these projects are not representative of the wider population of web-based Java applications or their tests. However, the projects that we selected demonstrate a wide range of testing practices: Jenkins topping in with $9,330$ tests, and Struts with only 15, showing that RIVULET can successfully find vulnerabilities even in projects with very few tests. We are quite interested in finding industrial collaborators so that we can apply RIVULET to proprietary applications as well, however, we do not have any such collaborators at this time.

Chapter 5

Detecting and Preventing ROP Attacks using Machine Learning on ARM

5.1 Overview

The Advanced RISC Machine (ARM) processor is a modern processor being widely used in many everyday devices such as smartphones, thermostats, refrigerators, and smartwatches. ARM claims that more than 200 billion ARM processors have been shipped by 2021[1]. Moreover, since ARM supports low power consumption without sacrificing performance, industry is shifting towards ARM processors, as observed in a comprehensive analysis of ARM and x86 processors by Gupta et. al (2021) [53]. Hence, due to the fast growth of mobile technologies and the internet-of-things (IoT) [125], ARM is becoming a more appealing target for control flow attacks aiming to acquire the capability to control a system. A popular method to that end is code injection, where an attacker exploits memory bugs as to maliciously altering the program's behavior or even taking full control over a system. Memory exploitation can be done by writing new machine code into the vulnerable program's memory or by reusing existing code. The latter is imperative when a protection technique known as W $\oplus$ X [1] is applied, which stipulates that memory is either writable or executable (but not both). Return-into-libc (RILC) [133] is a relatively simple code-reuse attack where a call stack is manipulated such that control is transferred to the beginning of an existing libc function, such as *system()*. For maximum expressiveness [57], return-oriented programming (ROP) [25] was introduced, which exploits a software vulnerability by chaining existing *gadgets* (small snippets of code ending in a return opcode) together in arbitrary ways. Moreover, Checkoway et. al. [30] show that ROP attacks can be mounted even without using return instructions, on both the x86 and ARM architectures.

To detect and protect against code-reuse attacks on ARM (e.g., ROP and its sibling *jump oriented programming* (JOP) [22]) some techniques have been proposed that try to enforce control flow integrity (CFI) via dynamic binary instrumentation (DBI) [65, 112] or the

[1] https://www.arm.com/blogs/blueprint/200bn-arm-chips

ARM CoreSight debugger [83, 84, 85, 103, 102], supplemented with *meta-data* collected by static analysis. Most of them rely on a *shadow call stack (SCS)* [111] for stateful backward edge protection (i.e, to detect ROP), while using a range of different static forward-edge policies such as *branch-table* (generated from CFG) and *branch regulation* (BR) [75] (i.e., to detect JOP). However, the techniques that use dynamic instrumentation suffer from high performance overhead while those that use the ARM CoreSight debugger suffer from high storage overhead. Besides, the hardware monitor that uses the hardware debugger could drop traces given a sufficiently high branch rate since the monitor requires more time to process a trace than the rate at which branches occur on the target processor [41]. Another limitation of using debugger traces to detect CRA attacks is that the hardware debugger can be used by an attacker to circumvent the security of the system. If the attacker can access the debug interface, he could use it to tamper with code and data memory, or even disable the hardware monitor by tampering with the tracing mechanism [41]. Therefore investigating whether another line of defense can detect ROP attacks on ARM accurately and precisely with low performance and storage overhead is advisable.

The ARM processor provides hundreds of hardware events related to instructions that can be monitored during process execution using hardware performance counters (HPC) [38]. On x86 there exist research that tries to detect ROP attacks by investigating the characteristics of hardware events during an attack using HPCs and machine learning [117, 43]. However, to the best of our knowledge there is no corresponding research for ARM. Hence it is important to investigate whether such a technique can detect code reuse attacks, such as ROP and JOP, on ARM-based applications.

In this thesis, we evaluate the suitability of a combination of HPC and machine learning techniques to detect and prevent ROP and JOP attacks on the ARM platform. Note that the myriad of HPC events on ARM differ from x86, as well as the execution model (e.g., there is no dedicated return opcode on ARM). The HPCs count the occurrence of certain hardware events on the ARM processor when executing a program, but it has not been investigated whether the events for normal and ROP attack executions differ significantly enough to enable automatic detection. Hence, we create a machine learning model of the behavior on ARM-based Raspberry Pi machines to address this question empirically.

Our machine learning approach computes models for *runtime monitoring*. The *offline training* examines several machine learning techniques and generates a set of classification models from HPC training data collected during benign executions and attacks. To that end, we developed a novel tracer that commences recording of HPC events in executions with ROP attacks only when this attack actually starts (i.e., the first gadget of the exploit executes), which improves the classifier's accuracy by 12% over recording the program's complete execution as previous work. The *runtime monitor* contains a *modified program loader*, a *kernel*

module and a *classifier*. The program loader configures the CPU using the tool *perf* as to track the set of HPCs required for the trained classifier, the kernel module computes the delta of these HPCs each time an interrupt occurs and feeds these values to the machine learning-based classifier, which labels the recent program execution as an attack or benign.

To obtain an optimal classification model our approach trains models using multiple machine learning approaches. Of eight machine learning techniques examined, the optimal classification model trained – SVM with a RBF kernel – displays a 92% and 91% accuracy for Raspberry Pi 4 and Pi 3, respectively. Leveraging this optimal classifier we evaluate ROP attack detection via runtime monitoring on Raspberry Pi using 15 exploits (based on four ROP attack variants) of real-world vulnerable applications. The detection of these attacks at runtime provides 75% accuracy, and we will elaborate on possible technical reasons for this difference.

Finally, we compared the detection accuracy of ROP vs JOP as well as Raspberry Pi 3 vs. Pi 4 (using dedicated models for each processor), which only yield insignificant differences when using dedicated models for each type. The latter is in sync with x86, where even switching inside the same processor family resulted in a significant decline of the detection rate [117].

5.2 Contributions

The major contributions of this thesis presented in this chapter revolve around investigating how well a promising line of defense against code-reuse attacks on the x86 platform transfers to the ARM platform:

- In order to evaluate how well control-flow attacks can be detected on the ARM platform using HPCs and machine learning techniques, we implemented a runtime monitor containing a modified program loader, kernel module and a classifier implemented in the kernel space to synchronize with the unmaskable kernel interrupts that trigger HPC reading.
- A novel debugger (tracer) that selectively records the actual attack section of a program subject to a control flow attack, in order to improve the classification model during the offline training.
- Compilation of a benchmark of 15 exploits (of four ROP variants) for the ARM platform (i.e., Raspberry Pi) from 8 real-world vulnerable applications. This benchmark is leveraged for offline training and online monitoring. Given existing ROP exploits are predominantly for x86 processors generating exploits for ARM processors is a complex, mostly manual task.
- A comparison of eight machine learning techniques to identify the optimal classification model.

- An evaluation of the ROP attack detection's accuracy and performance overhead considering various evaluation criteria.

5.3 Methodology

The methodology we use to detect and prevent ROP attacks on ARM platforms is based on learning the behavior of micro-architectural events in the CPU, combining HPC readings and machine learning techniques. Figure 5.1 shows our approach to detect and prevent ROP attacks. It starts with creating exploits for different real-world vulnerable applications on the ARM platform. Subsequently, we collect and pre-process the required training data by profiling both ROP attack and benign executions and reading the HPC values using the kernel's *perf* event profiler. Next, we apply machine learning techniques to train a set of models that detect and prevent ROP attacks in terms of binary classification of the programs' executions into *benign* or *under attack/ROP* based on the data collected from the HPCs. The offline training phase determines a combination of events that characterize ROP and, benign executions. After finding the model that provides the best combination of HPC events for attack detection, we use our online monitoring technique to detect and prevent the ROP attacks in a real-time execution setting.

At a high level we re-evaluate the approach of HadROP [117] and EigenROP [43] in the sense that we also leverage HPCs and machine learning to detect ROP attacks. However, we are investigating this approach's applicability to the ARM platform (with its growing prevalence) due to its peculiar instruction set (no explicit *ret* instruction) and dedicated set of HPC events. Moreover, we train several machine learning classifiers and choose the optimal algorithm based on its performance. In addition, we selectively record the actual micro-architectural events of the ROP attack section only, rather than of the entire ROP attack execution, which increases our classifier's accuracy from 80% to 92%. Finally, unlike EigenROP our machine learning approach follows supervised learning and the online monitoring section also differs from HadROP's approach in its program loader and classier implementation. All the steps of our approach are explained in detail below.

5.3.1 ROP Exploit Creation on ARM

In this section, we present how to create ROP exploits for vulnerable *real-world applications* on the ARM platform, as a sufficient set of ROP-affected malicious executions is required to train a stable classifier. Although ROP attacks on ARM are not a new idea, exploits of real-world vulnerable applications are hard to find. The first challenge entails installing and compiling vulnerable applications originally developed for x86 on the ARM platform and reproducing exploits known from x86, i.e. finding proper ARM gadgets and chaining these gadgets together into an ARM-specific exploit. The exiting ROP detection approaches [83, 84, 85, 103, 102] for ARM use not more than five (3 ROP and 2 JOP) simple example attacks

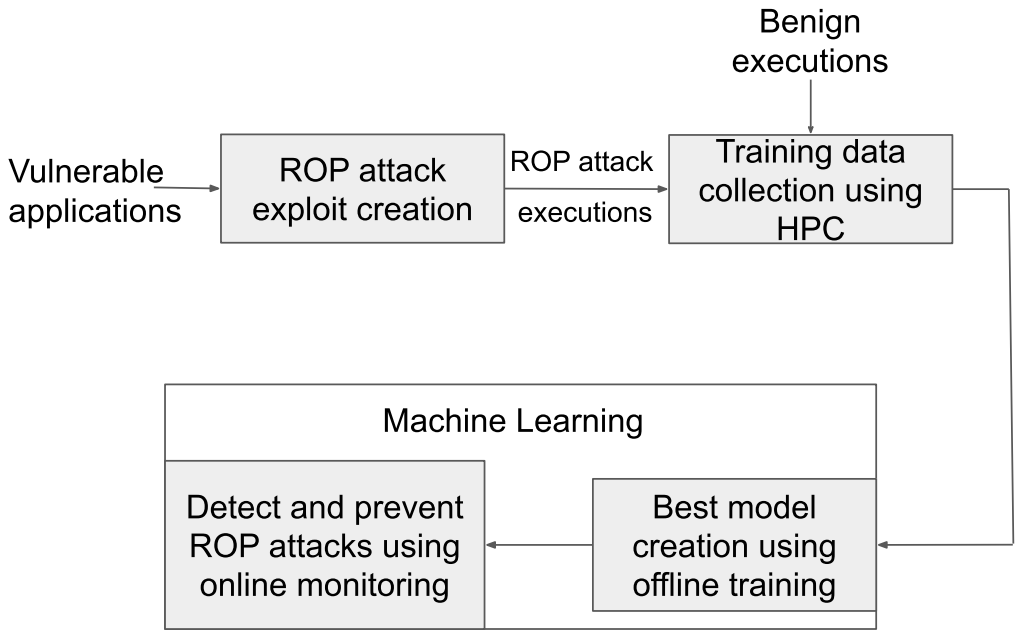


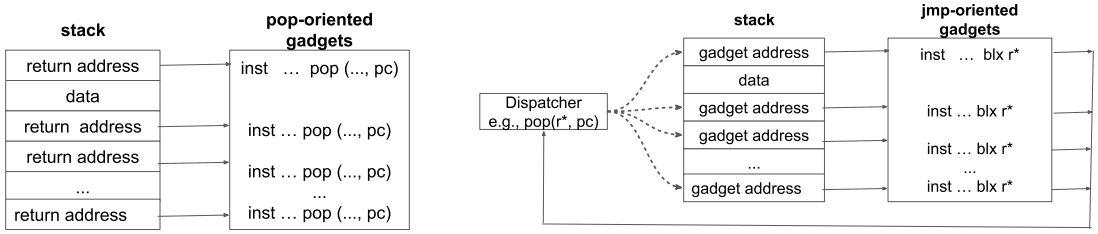
FIGURE 5.1: Our approach to detect and prevent ROP attacks

based on shellcodes provided by shell-storm [2], which exploit programs that are by no means real-world.

In general, the exploit creation process for ARM is challenging as we need to customize an available ROP attack for a different platform. Hence, we first need to locate the vulnerable code in the program, i.e., the size of the buffer vulnerable to a buffer overflow attack which determines the position of the return address on the stack succeeding the buffer. Then, we use ROPgadget to identify gadgets that can be chained to perform the ROP attack types we need. Finally, we create the ROP exploit (payload) by exhausting the buffer with random values (e.g, "AAA...") until we reach the return address and overwriting the return address with the address of the first gadget. The subsequent values (usually return addresses of further gadgets) must be carefully selected such that the gadgets are chained in the order given by ROPgadget to accomplish the intended attack.

Moreover, to increase the diversity of the training and testing data set, we implemented several ROP attack variants (Ret2ZP [66], JOP [22], Ret2mP [160] and Stack pivoting [118]), which, based on the types of gadgets used to create gadgets chains, can be generalized into ROP (gadget chains ending in POP) and JOP (gadget chains ending in BLX) attacks. More generally, Figure 5.2 shows how we modeled the ROP and JOP attack exploits creation on ARM platforms using *pop-based* and *blx-based* gadgets. In ROP, gadget addresses are loaded into the program counter (PC) register using POP. In JOP, control flow (CF) is driven using

[2] http://www.shell-storm.org/



(A) ROP model

(B) JOP model

FIGURE 5.2: ROP vs JOP

a special dispatcher gadget that executes the gadget chain. A register that points to the gadget address list is used as the virtual program counter. In both variants, to provide arguments to a function, the contents of function argument registers (i.e., r0-r3) must be assigned before CF is redirected to the desired function. For instance, if we want to open the system's shell the register r0 must point to the address of *"/bin/sh"* before CF is directed to the address of the *system* function. Overall, we developed 15 exploits (8 ROP and 7 JOP) attacking 8 real-world vulnerable applications.

5.3.2 Data Collection on Arm using HPCs

Data collection on the ARM processor is a most challenging process since there is no tool available that can directly and continuously collect and store the relevant data separately from the program to be executed. So we had to modify existing profiling tools such as *perf* and the Linux kernel module on ARM to enable the recording of HPC data. Furthermore, we developed a program that traces the ROP program and records only the actual ROP section, i.e., starts just before the execution of the first ROP gadget.

To record HPC data via *perf* the interrupt handlers of the *performance monitoring unit* (PMU) in the ARM processor must be modified. The interrupt handlers then regularly poll the HPC counters, which contain the frequencies of the hardware events since the previous interrupt, as attributes for model training. We leverage the kernel message log (*printk()*) to store that data for offline training.

Tracer tool to record only the ROP part

During a ROP attack, the stack is overwritten by the adversary with a chain of gadgets pointing to existing executable binary code. However, the program execution exhibits regular behavior until the first gadget is invoked, as the actual ROP attack, which manipulates the program's control flow, starts at that time. To gather HPC data only for the actual ROP process, we implemented a tracer tool that reads configuration data from a file, including the first gadget's address (to set a breakpoint), the path to the vulnerable application and the ROP attack's payload.

The tracer acts like a debugger, injecting a trap instruction via the *ptrace* API, which suspends the target process at the beginning of a ROP chain. In summary, the tracer performs the following steps to exclusively record the ROP behavior: First, it suspends the program and replaces it with a forked child process. Then, it injects the trap (i.e., a synchronous interrupt caused by an exceptional condition, in our case a breakpoint) and runs the vulnerable program until the breakpoint is reached, indicating that the actual ROP execution is about to begin. Thus *perf record* is triggered to record the HPC data for the remainder of the program's execution.

In order to virtualize the HPC readings to each process and thus remove noise from concurrently executing processes we apply the *-p* option to *perf record*. However since only a certain number of HPC events can be recorded at any time, we sample them in smaller batches and train the machine learning algorithms with the combined data in order to select the most relevant HPC event types (features) for ROP detection.

5.3.3 Offline Learning using HPC Data

The offline model learning phase on a high level follows the approach of HadROP [117]. Apart from considering ARM instead of x86, we are also evaluating a number of machine learning techniques beyond a SVM. Training data is gathered by recording the HPC features based on a given set of ROP attacks and benign program executions. Using feature selection techniques we derive the best classification model for online ROP attack detection via the learned classifiers. To determine the models, we use several machine learning (ML) techniques that classify the feature vectors into malicious (ROP attack) and benign. The feature vectors comprise the deltas of the HPC counts per sampling period of a particular program run. The sampling period needs to be fixed before the collection of training data starts and cannot be changed subsequently.

The model generated using the offline training is expected to contain a small subset of a large number of available HPC events by applying feature selection techniques that select the most meaningful HPC events for ROP detection. The problem with feature selection is that the measurements of HPC events during the data collection phase are noisy for several reasons: The complexity of the CPU and non-determinism of the HPC specification renders reproducing HPC events across multiple sub-sampling runs difficult. Moreover, context switching might trigger additional events since HPCs are saved to the process control block.

Despite this problem, the feature selection technique helps determine a small subset of HPC event types supported by Raspberry Pi that best matches the expected results from an in principle large number of events (in our case 51). Note that feature selection is imperative as the selected number of event types must also adhere to the constraints of the target CPU. For instance, on Raspberry Pi, *perf* cannot sample more than 8 event types simultaneously. To resolve this issue we sample using subset batches of the events and

combine them (synchronized according to time) for offline learning and feature selection in order to determine the most suitable combination of HPC events.

Model Selection Methods: We train eight ML classifiers by providing two sets of feature vectors, collected from ROP and benign program runs. Moreover, parameters like an error penalty C, which allows more or fewer mis-classifications, are given as input parameters. Choosing the appropriate parameters that result in an optimal model is not trivial. Hence, we select the optimal parameters through a dynamic oscillating search [137].

Our model selection approach uses *k-fold* cross-validation to optimize the selection. Cross-validation is a statistical method re-sampling procedure to evaluate and compare machine learning algorithms by splitting data into two segments: one segment for training the machine learning model, and the other segment to validate it. Typically, the training and validation data sets must cross-over in successive rounds such that each data point can be validated against. These evaluation results guide the dynamic oscillating search, which determines the next set of potentially optimal features and parameters. Iterating this process results in optimal parameters, e.g., for SVM in an error penalty and optimal hyper-plane, based on a subset of HPC event types of appropriate size. Considering the bias-variance trade-off in k-fold cross-validation, we choose $k = 10$ as the model selection method as recommended by Kohavi et al. [81].

5.3.4 Online Kernel Monitor

Figure 5.3 presents our online monitoring process using the machine learning model, which consists of a modified *program loader*, a *kernel module* and a *classifier*. The program loader configures the CPU using Linux's *perf* tool to track the set of HPCs that are relevant for the trained model to classify an execution as ROP or benign. Moreover, it notifies the CPU to raise an interrupt every N clock cycles. At each interrupt the kernel module computes the deltas of the HPC count values and feeds those to the classifier, which is implemented in the kernel space to synchronize with the readings of the HPCs during each interrupt. Whenever the classifier determines that a ROP attack behavior occurred, the process can be suspended or other defensive actions, such as notifying security personnel, can be taken. As HPCs are updated in hardware, the performance overhead of online monitoring stems only from handling the interrupt, reading the counters, and evaluating the classifier.

Program Loader

We modified the routine that starts a program (program loader). Concretely, we modified the *_libc_start_main* function, which calls the main function of a program, to configure the HPCs selected for classification. In particular, we are adding the *perf record* command, which samples these HPC events, before the call to the main function of the program. The *perf record* command uses the *frequency* N and selected *features* (HPC events) of the optimal

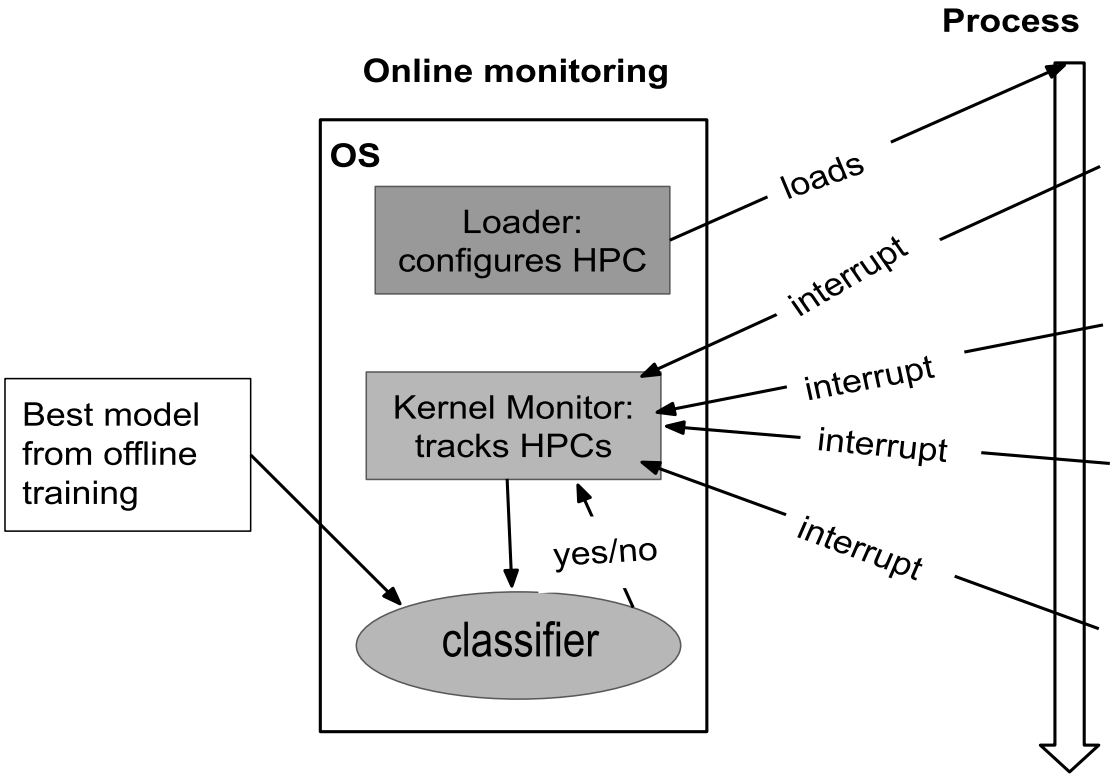


FIGURE 5.3: Our online monitoring approach to detect and prevent ROP attacks

classification model as input in addition to the target program to be runtime monitored. To use this modified program loader we leverage the environmental variable *LD_PRELOAD*. *LD_PRELOAD* contains one or more paths to shared libraries, or shared objects, that will be loaded before any other shared library including the C runtime library (libc.so). However, in order not to end up calling the modified loader recursively we need to unset *LD_PRELOAD* after configuring the HPCs.

Kernel module

The kernel module contains a modification of perf's interrupt handler to recognize the configuration specific to the classifier. Note that interrupts produced by HPCs are recognized as non-maskable interrupts, which must be handled by the kernel and cannot be ignored (masked). The interrupt handler extracts the delta (change of counts) readings of the selected[3] HPC events at each interrupt and passes it as an array to the monitor that contains the classifier, which in turn performs the classification and redacts the execution based on the output of the classifier.

[3]HPC events selected by feature selection to provide the optimal classification model

The classifier

During offline training, several machine learning techniques are evaluated (see section 5.4). Given that the SVM provides the best classification model we use that model for online monitoring. LibSVM determined the best HPC events that characterize ROP attacks during offline training. However, we cannot directly use LibSVM for online monitoring since the kernel module, which extracts the HPC readings during the interrupt, is not correctly synchronized with user space, which would call LibSVM's *svm-predict* function. Due to the lacking synchronization most of the HPC readings would be missed due to delays in the user space actions such as extracting the model file and performing the SVM prediction calculations. Hence, we decided to implement the SVM prediction (classifier) directly in kernel space. Yet, there are two problems with implementing the SVM's prediction in the kernel module.

1. Reading the SVM model file in the non-maskable interrupt context
2. Using floating point arithmetic in the kernel module

The kernel space does not support reading files in the interrupt context. Hence we cannot read the relevant classification inputs directly from the optimal classification model file. Since this data is static, we extract the relevant data from the model file and store it into arrays and/or variables before online monitoring.

Moreover, the kernel module is unable to support floating point arithmetics [21, ch. 5], which prohibits implementing the SVM prediction formula directly. Hence, we opted for solving this problem using *fixed-point arithmetic*[4]. For instance, considering the SVM prediction formula for the RBF kernel given below, we need to use fixed-point arithmetic to represent e, γ and the support vector coefficients (a_i), which usually have floating point values. The bound nSV in the formula represents the number of support vectors.

$$y = \sum_{1}^{nSV} a_i * e^{-\gamma * |X - X_i|} + b$$

With both of these issues solved the classifier can now receive HPC readings from the interrupt handler and predict the class of the program execution. Based on the output of the classifier the online monitor can then decide what to do with the running application. If the behavior of a ROP attack is detected the monitor can suspend the process or notify the responsible body (e.g. security personnel) about the issue.

[4] https://en.wikipedia.org/wiki/Fixed-point_arithmetic

TABLE 5.1: Accuracy evaluation of ROP attacks using different frequencies on Raspberry Pi 3 and Pi 4. We also show the optimal C and γ that provide the best accuracy for each frequency.

Frequencies	Pi 3 model B			Pi 4 Model B		
	Cost (C)	**Gamma (γ)**	**Accuracy**	**Cost (C)**	**Gamma (γ)**	**Accuracy**
3000	4	0.000031	0.89	4	0.000031	0.87
4000	64	0.000488	0.91	256	0.000488	0.91
5000	4	0.000122	0.85	256	0.000488	0.89
6000	4	0.000031	0.80	256	0.000488	0.88
7000	16	0.000031	0.82	64	0.000488	0.87
8000	256	0.000031	0.82	256	0.000122	0.84
9000	4	0.000031	0.82	256	0.000031	0.92

TABLE 5.2: Precision, recall and accuracy evaluation of ROP attacks using different machine learning techniques for Raspberry Pi 3 and Pi 4.

Model Name	Pi 3 model B (4000Hz)			Pi 4 Model B (4000Hz)			Pi 4 Model B (9000Hz)		
	Precision	**Recall**	**Accuracy**	**Precision**	**Recall**	**Accuracy**	**Precision**	**Recall**	**Accuracy**
KNeighbors	0.89	0.87	0.89	0.72	0.81	0.78	0.87	0.95	0.91
AdaBoost	0.85	0.96	0.90	0.80	0.97	0.89	0.84	0.90	0.88
GradientBoosting	0.88	0.96	0.92	0.88	0.97	0.92	0.86	0.93	0.90
DecisionTree	0.82	0.96	0.88	0.85	0.97	0.91	0.87	0.83	0.85
RandomForest	0.83	0.96	0.89	0.85	0.94	0.90	0.88	0.90	0.90
ExtraTrees	0.85	0.96	0.90	0.85	0.97	0.92	0.87	0.96	0.92
LinearDiscriminant	0.70	0.98	0.79	0.72	0.98	0.84	0.85	0.98	0.92
SVM RBF	0.87	0.98	0.91	0.82	0.98	0.90	0.85	0.98	0.92

5.4 Evaluation

Our evaluation environment consists of Raspberry Pi 4 Model B and Raspberry Pi 3 Model B with kernel version 5.4. The evaluation comprises multiple experiments to obtain the parameters that provide an optimal model of ROP attack classification, to measure the accuracy of ROP attack detection and of the performance overhead of online monitoring.

5.4.1 Optimal Model Selection and Accuracy of Offline Training

To find the optimal model during offline training, we perform many experiments with varying sampling frequencies, several machine learning techniques and their classification parameters on both Raspberry Pi 3 and Pi 4.

Accuracy evaluation with respect to different frequencies

In order to reduce the space for the sampling rate we conducted initial experiments with only one ML technique [117]. Table 5.1 shows the classification accuracy of SVM models for different frequencies. The cost (C) and gamma (γ) parameters that provide the best accuracy for their respective frequencies are also provided. The experiments use the crossover value $k = 10$. The result of this evaluation shows that data collected by HPC recording with a frequency of 4,000 Hz on Raspberry Pi 3 provides the best accuracy, which is 91%, and on Raspberry Pi 4 data collected by recording with a frequency of 9,000 Hz provides the

best, which is 92%, but the recording with 4,000 Hz on Raspberry Pi 4 also provides similar results, i.e., it is 91%. However, this does not mean we always get the same results each time we collect data with these frequencies due to the non-deterministic behavior of HPC readings. The HPC readings vary at each execution for the same application even with the same frequency. Although the frequencies that provide the best model vary, we observe that there is no significant difference between Raspberry Pi 3 and Pi 4 in the offline classification accuracy.

Accuracy evaluation with respect to different machine learning techniques

With the reduced set of sampling rates, we then evaluate our offline training with respect to multiple machine learning techniques, the result of which is provided in Table 5.2. In addition to the accuracy, we also evaluate the recall and precision of the models obtained using the frequencies 4,000 Hz on Raspberry Pi 3, and 4,000 Hz and 9,000 Hz on Raspberry Pi 4. Note that *recall* indicates the percentage of feature vectors (FV) correctly classified as ROP attack of all FVs of ROP attacks. Similarly *precision* indicates the percentage of the FVs *correctly* predicted as ROP attack in all FVs classified as ROP. So both the recall and precision provided in Table 5.2 are with respect to the ROP attacks. In contrast, accuracy measures the overall classification accuracy for both ROP and benign executions. The training data used in all experiments is also balanced, i.e., we use the same number of ROP and normal feature instances as input to the machine learning techniques.

As we observe from the result, our evaluation indicates that the detection accuracy of the optimal model using an SVM kernel is at least in the top two. For instance, considering the Raspberry Pi 4 ExtraTrees, LinearDiscriminant and SVM RBF kernel provide the best accuracy which is 92%. However if we also take the recall both LinearDiscriminant and SVM RBF kernel have a higher value than ExtraTrees though somehow less in precision. Similarly if we consider Raspberry Pi 3 GradientBoosting provides the best accuracy of 92% whereas SVM RBF kernel provides 91%. However the SVM RBF kernel provides better recall than GradientBoosting with almost similar precision. From these results on both Raspberry Pi 3 and Pi 4 we conclude that the model obtained using the SVM RBF kernel is optimal for online monitoring. All subsequent experiments are thus based on a SVM with RBF kernel only.

Accuracy evaluation with respect to ROP and JOP

We further evaluated the classification accuracy between ROP and JOP attacks to understand how different attack types affect the behavior of the HPC values. Table 5.3 provides the accuracy of both ROP and JOP attacks using different frequencies on both Raspberry Pi 3 and Pi 4. On Raspberry Pi 3 the models from ROP attacks yield higher accuracy than from JOP attacks for frequencies 3 kHz and 5 kHz. In contrast, considering 8 kHz and

TABLE 5.3: Accuracy evaluation of ROP vs JOP attacks using different frequencies on Pi3 and Pi4.

Frequency	Pi3 model B		Pi4 Model B	
	ROP	**JOP**	**ROP**	**JOP**
3000	0.89	0.85	0.85	0.90
4000	0.86	0.87	0.89	0.82
5000	0.84	0.81	0.85	0.84
6000	0.83	0.87	0.80	0.85
7000	0.81	0.82	0.83	0.82
8000	0.81	0.85	0.82	0.84
9000	0.85	0.85	0.85	0.87

6 kHz the model from JOP attacks has higher accuracy than from ROP attacks. For the other frequencies the accuracy is similar for both variants. Similarly on Raspberry Pi 4 the model from ROP attacks provided higher accuracy than from JOP attacks for 4 kHz but for 3 kHz and 6 kHz frequencies the model from JOP provided higher accuracy than the model from ROP attacks. For the other frequencies there is not much difference. These results indicate that we can not generalize that one attack type is more detectable than the other, i.e., it depends on the frequencies and even may vary when recording it again even with the same frequency.

HPC events providing the best classification model

As we observe from the offline training evaluation, we got a classification accuracy of 92% on Pi 4 and 91% on Pi 3. The selected HPC events that provide these results for the Pi 4 are: *branch-misses*, *branch-load-misses*, *ld_spec*. Similarly, the selected HPC events for Pi 3 are: *cache-misses*, *branch-misses*, *cid_write_retired/*. It is interesting to note that the occurrence of ROP attacks impact the count of these three HPC events but significantly of the others, as our feature selection process could have chosen up to eight HPCs for simultaneous tracking. These HPC events will also be used for online monitoring, i.e., to detect and prevent the ROP attacks.

5.4.2 Accuracy of Online Monitoring

To evaluate the detection and prevention accuracy of our online monitoring, we used three real-world vulnerable applications (php, dnstracer, mcyrpt), which were not used for training the model. Since we have benign as well as both ROP and JOP attacks for these applications, we have a total of 9 tests, out of which 7 are consistently detected correctly, i.e., the ROP and JOP attacks in *mcyrpt* go undetected more often than not. Note that since HPC values are nondeterministic those detected at one time may not be detected another time and vice-versa. Moreover, since the HPC recording during an execution of a program provides many feature instances there is a high possibility that instances are predicted

TABLE 5.4: Performance overhead evaluation of real-world applications on Pi 3 and Pi 4.

Application	vulnerability	slowdown Pi 3	slowdown Pi 4
Crashmail 1.6		8.7%	4.6%
Pms 0.42		11%	9%
Php 5.3.5	CVE-2011-1938	4%	11%
Netperf 2.6.0		7.6%	9%
Wifirix		5.4%	5%
Dnstracer 1.8.1	CVE-2017-9430	5%	4%
Mcrypt 2.6.8	CVE-2012-4409	3%	2.6%
Nethack 3.4.0	CVE-2012-4409	5.6%	4.3%

wrongly during the attack and benign executions. To minimize this, we have to look for the optimal maximum number that instances are predicted 1 (ROP attack) consecutively to determine that a real attack has started. In our case we used 10, i.e, if 10 consecutive instances are predicted as 1 we assume there is an attack and the program execution will be suspended, otherwise we assume that a false prediction of the instances has occurred and consider the execution as benign. Hence the ROP and JOP attacks of mcyrpt are being mostly undetected probably since their exploit has small gadget chains relative to the others. In general, we have also tested our online monitoring with small hand-crafted ROP attack examples and the detection accuracy of our online monitoring is around 75%, on average.

5.4.3 Performance Overhead of the Online Monitoring

The performance overhead of runtime monitoring is measured by comparing the time of execution for the ROP attacks with and without the usage of the runtime monitor. Table 5.4 shows the performance overhead (slowdown in %) of 8 real-world vulnerable applications on Raspberry Pi 3 and Pi 4. For most of them, the slowdown is higher on Raspberry Pi 3 than in Pi 4 but for Php 5.3.5 and Netperf 2.6.0 we got a higher overhead on Raspberry Pi 4 than on Raspberry Pi 3. However, the execution time of both applications is still smaller in the Raspberry Pi 4 than on Raspberry Pi 3 if we consider it separately even with the application of the online monitor. For instance, the execution time for Php 5.3.5 on Raspberry Pi 3 is 1.5s whereas on Raspberry Pi 4 it is 0.5s with the application of the runtime monitor. In general, the overhead evaluation using these 8 applications shows that our implementation of the online monitoring provides a slowdown in the range of 2.6% –11%. On average the slowdown is 6.3%, on Raspberry Pi 3, and 6.2% on Raspberry Pi 4, indicating that the performance overhead is almost identical on both.

Chapter 6

Detecting ROP Attacks on Firmware-Only Embedded Devices Using HPCs

6.1 Overview

The Internet of Things (IoT) and embedded devices make use of dedicated commercial off-the-shelf microprocessors and many of these devices largely depend on firmware with a C-language codebase and software development kit [6]. As expected, the battle of wits between the engineers creating offensive and those inventing defensive strategies is a never-ending one, and attackers are increasingly and desperately bug hunting these devices using memory corruption techniques such as buffer overflows and heap corruption, which are common in C applications, to find a way to evade device security. Frequently, attackers technically use an approach called *Return-Oriented Programming* (ROP) [30], one of the most dangerous security exploit techniques to take advantage of software weaknesses such as buffer overruns (e.g., in the C language), overwriting the call stack, and gaining control over the program's control flow [164]. Omitting the need to inject malicious binary code, the attackers meticulously select and execute multiple tiny sequences of machine instructions (called *gadgets*) that are already in memory [134, 44]. The attacker will construct a payload based on the addresses of the selected gadgets and corrupt the stack such that the return address of the topmost stack frame points to the first gadget. Since each ROP gadget ends in a return instruction, gadgets can be chained together to build complex exploits by ensuring that the next return address on the stack points to the succeeding gadget. The major challenge to preventing ROP is that the gadget's instructions are located in the executable memory area of the original program and therefore ROP can circumvent mitigation mechanisms such as *data execution prevention* and coarse-grained *address space layout randomization*. A popular variant of ROP is *Jump Oriented Programming* (JOP) that

involves chaining gadgets ending in a jump instruction and controlling the control flow via a special gadget called the *dispatcher gadget* [23].

Most of the existing ROP countermeasure techniques focus on x86 or other architectures e.g in [40, 64, 94, 42], processors for embedded devices like the Xtensa core have only been investigated rudimentary. Xtensa is a Tensilica processor platform manufactured by Cadence®, with highly customizable and configurable processors that found wide application in HiFi audio and voice digital signal processors. The Tensilica core family includes the Xtensa LX and NX processors, and different versions of the core have been adopted by vendors like Microsoft, AMD, and Espressif [159, 171]. Millions of IoT devices (including industrial IoT devices, e.g., eModGATE[1] and Moduino X Series[2]) and embedded systems using ESP 32 (LX6) and its predecessor ESP 8266 (LX106)—which are economical and low-power systems on a chip—are based on Xtensa core. Firmware-only devices are characterized by deterministic interrupt-driven tasks due to a lack of a scheduler (no OS). The firmware is typically stored in rewritable, nonvolatile memory (flash), without fine-grain privilege separation and execution isolation available in a conventional OS [34]. Usually, they are supported by manufacturer header files, and the absence of third parties drivers/firmware is believed to add trust and control [59]. However, this restricted the use of custom security, and being resource-constrained(e.g low memory/storage), it is challenging to deploy a sophisticated solution against memory corruption attacks on them.

Motivation: First, ROP attacks on Xtensa are *not well documented*, even though the chips are present in almost every WiFi-based home automation device. However, there exist some records of buffer overflow vulnerabilities exploitation leading to ROP. For example, on Expressif's ESP8266 (with Xtensa LX106 core) based on FreeRTOS in Expressif's IoT development framework, the attack has resulted in bypassing network credentials and making the devices perform unintended operations [156]. A similar memory exploitation attack with the *common vulnerabilities and exposure* number *CVE-2019-12588* has also been used to crash Xtensa (LX106 and LX6) WiFi devices causing a denial of service to legitimate users, to name but a few [50].

Second, micro-controller devices are often resource-constrained but there are very few studies on ROP in IoT platforms without operating systems but firmware, that runs directly from flash memory. Likewise, almost all of the previous works on ROP detection using HPCs have primarily *considered devices with a full-blown configuration*, where it is possible to inspect the usual suspects – several cache level events, return misses directly along with all other available hardware events [117, 37] that are usually either not available or inapplicable in embedded platforms. Also, the most recent low-level detection study on

[1] `https://iot-industrial-devices.com/category/esp32/`
[2] `https://moduino.techbase.eu/`

micro-controller we found proposed the use of control flow integrity but they did not include any implementation or result to support the feasibility of the approach [94]. In contrast, we investigate an alternative cost-effective approach using HPC events on a low-configuration Xtensa processor.

These shortcomings, together with the danger that IoT devices may not receive security updates as frequently and lastingly as personal computers or mobile devices, serve as the basis for this research.

This thesis presents the first detection of ROP, and its variant *Jump-oriented programming* (JOP), in a firmware-only environment combining machine learning predictive capability with the HPCs. Our approach depends on the variations in the HPC micro-architectural events triggered by ROP and normal program execution. We implemented attack scenarios using instrumented programs and exploits that perform operations similar to those in known microprocessor benchmark programs. Recorded micro-architectural events are used to train machine learning binary classifiers. The learned model identifies relevant HPCs, which could serve as predictors of ROP/JOP execution even in configurations where features non-typical to conventional processors, like instruction memory and data memory, are available. we also evaluate our approach and the results indicate a high precision, recall, and accuracy of the classifier predictions.

6.2 Contributions

The major contributions of this thesis covered in this chapter are:

- Show how ROP and JOP attacks could be orchestrated on Xtensa processors
- Present the first practical work on detecting ROP and JOP attacks in a firmware-only embedded system using HPC and Machine learning.
- Identify HPC events that distinguish ROP and benign program executions on Xtensa.

6.3 Threat model:

Our focus is on preventing attacks that exploit memory corruption in firmware or application functions through buffer flows and that result in a ROP or JOP execution on bare-metal Xtensa. We assume that the device has limited hardware functionalities but is protected using a machine-learning classifier based on the HPC that we present. The attacker is also assumed to be able to access the device either physically or through a network connection. Since we are interested in the attack that takes place when executing firmware from the flash memory, the attacker uses a payload to execute gadget instructions that are at the static address range. In practice, gadgets in the Xtensa Boot ROM are mapped to the static range, irrespective of the platform being used. When an attacker exploits vulnerable firmware functions, our approach is aimed at feeding the captured HPC events from the attacker's execution to the machine-learned classifier to predict ROP or JOP behavior.

6.4 Methodology

In this section, we demonstrate how the ROP and JOP attacks work on Firmware-Only embedded devices that use Xtensa processors and provide the first insight approach to detect these attacks using HPCs and machine learning.

6.4.1 Xtensa Assemblies and Gadgets

Xtensa ABI Assemblies

In section 2.8 the background of Xtensa architecture is explained and we see that Xtensa supports two application binary interfaces (ABIs): *Call0* ABI and the *Registered Windowed* ABI. However, in this thesis, our principal target is the Call0 ABI processor configuration, which has been hit by some memory corruption attacks in recent years. We, therefore, briefly present the Call0 ABI assembly to demonstrate ROP and JOP attacks on this ABI configuration to provide a foundational understanding of Xtensa architectural behavior. To this end, we leverage the configurability feature of the Xtensa processor. We created, built, and installed a configuration for CALL0 ABI using Xtensa Xplorer version 8.0.10.3000 running on Windows 10. Xtensa Xplorer's Integrated Development Environment is based on Eclipse and comes with a pre-installed Software Development Kit for processor configurations and programming. We then compiled a simple *helloworld.c* program to illustrate the call0ABI assembly. The corresponding assembly is shown in Listing 6.1.

It is worth noting that some instructions in the Call0ABI use the *.n* suffix, which is the Xtensa processors' optional code density feature that provides 16-bit versions of some commonly used instructions. Technically, the compiler and the assembler use narrow instructions where possible to achieve better code density [28]. In `line 1`, the stack is decremented by 16 bytes (space allocation), and in `line 2`, the return address is saved to the top of the stack `*(a1+0)`. At the end of the assembly, the reverse is done before **`ret.n`**, i.e. the return address is restored into `a0` in `line 7`, and the stack is incremented by 16 bytes (space deallocation) in `line 8`. These two steps are similar in almost all Xtensa assemblies and will serve as the basis for chaining gadgets.

The Xtensa assembly language opcodes used throughout this thesis will be limited to a small subset of the entire Xtensa instruction set. Our interest covers mainly the *return* (**`ret`**), *jump* (**`jx`**), *call* (**`callx`**), *load* (**`l32i`** and **`l32r`**), *store* (**`s32i`**), *move* (**`mov`**), *add* (**`add`**), and *subtract* instructions (**`sub`**).

Xtensa Gadgets

ROP and JOP gadgets in Xtensa usually end with a **`ret`** and **`jx`** instructions respectively,

```
1 main:
2 60000bcc:   addi a1, a1, -16
3 60000bcf:   s32i.n a0, a1, 0
4 60000bd1:   l32r a2, 60000904 (600007b8 <_clib_rodata_end>)
5 60000bd4:   call0 60000c0c <printf>
6 60000bd7:   movi.n a2, 0
7 60000bd9:   l32i.n a0, a1, 0
8 60000bdb:   addi a1, a1, 16
9 60000bde:   ret.n
10
```

LISTING 6.1: Call0 ABI assembly

```
1 6000a383: 1148    l32i.n  a12, a1, 4
2 6000a385: 4149    l32i.n  a13, a1, 8
3 6000a387: 4128    l32i.n  a14, a1, 12
4 6000a389: 3108    l32i.n  a0, a1, 0
5 6000a38b: 30c112  addi    a1, a1, 16
6 6000a38e: f00d    ret.n
7
```

LISTING 6.2: Return gadget

although it is also possible to use codes ending with an indirect `callx` and `branch` instructions to an address stored in a register. For gadgets discovery, we extended the *xrop* tool[3] to extract and return valid Xtensa gadgets ending with the preferred instruction types. From our programs we extract and design Turing complete gadgets that perform data movement, arithmetic operations, branching, and function calls. The programs are compiled with Xtensa Xplorer, which outputs executable and linkable format binaries supported by the Xtensa LX7 board. Assuming the stack has been overwritten and preloaded with attacker-controlled addresses and values, it is, e.g., possible to use the *return gadget* in Listing 6.2 to load arbitrary values from the stack at `a1+4`, `a1+8` and `a1+12` into the registers `a12`, `a13`, and `a14` respectively before returning to the designated address.

Xtensa gadgets support very limited direct operations on memory addresses. Therefore, gadgets' addresses must either be loaded into a register from another register or the stack. One of the most common instructions in Xtensa binaries allowing direct memory operands is `l32r`, which can directly load the address of a string literal from memory. Thus, gadgets used for exploits are usually longer and with more side effects on registers, when compared to ARM. For instance, an equivalent of the gadget in Listing 6.2 in ARM could be as short as:

```
0x00010578 : pop(r3, r4, r5, pc)
```

[3] https://github.com/jsandin/xrop

```
1 60000210: 0338        l32i.n  a3, a1, 0
2 60000212: 7149        s32i.n  a4, a1, 28
3 60000214: 0003a0      jx  a3
4
```

LISTING 6.3: Jump gadget

```
1 60000e01: 4108        l32i.n  a0, a1, 16
2 60000e03: 0e4d        mov.n a4, a14
3 60000e05: 0000c0      callx0  a0
```

LISTING 6.4: Call gadget

An example of a *jump gadget*, which loads the content of the address stored in `a1+0` into `a3` and jumps to it is given in Listing 6.3.

Other types of instructions that can be used as gadgets are the *branch gadgets* and the *call gadgets*, an example of the latter, which performs an indirect call operation to a subroutine address in a register is shown in Listing 6.4. The `l32i` instruction loads an address from the stack at `a1+16` into `a0` and then jumps via a procedure call to that address.

6.4.2 Xtensa ROP Attack Process

Usually, more than one gadget is required to perform a complex exploit, the general process of chaining ROP gadgets is shown in Figure 6.1, which represents a stack that grows downwards (the buffer grows in the opposite direction) from a higher memory address to a lower memory address. The figure has two sections – the code section (memory region that is non-writable but executable), and the stack section (memory region that is writable but not executable). The code section contains the executable gadgets while the stack section stores the addresses of these gadgets (plus potential values to be read into registers by gadgets). Since we are exploiting a buffer overflow vulnerability to hijack the program's control flow via ROP, the gadgets' addresses must be placed behind the buffer starting at the current stack frame's return address. Let us assume that we have a function that is not performing a bounds check and that the function is using an unsafe C function like *gets* or *strcpy* to initialize a 16 bytes buffer variable. We can exploit this function by feeding it a payload that overflows the buffer and writes to the stack the addresses of *gadget1* (@address_of_gadget1), *gadget2* (@address_of_gadget2), and *gadget3* (@address_of_gadget3) respectively. In this example, these addresses will be 20 bytes, 24 bytes, and 28 bytes, respectively, from the beginning of the buffer. More importantly, the address of the first gadget (@address_of_gadget1) overwrites the return address in `a0` and the stack pointer `a1` becomes the gadgets counter i.e the number of times `a1` is incremented is equivalent to the number of gadgets executed. This whole process is depicted in Figure 6.1. Each gadget executes, increments the stack pointer, and returns to the address on the top of the stack until all the gadgets in the ROP payload have been executed. The order of execution of the instructions after the control

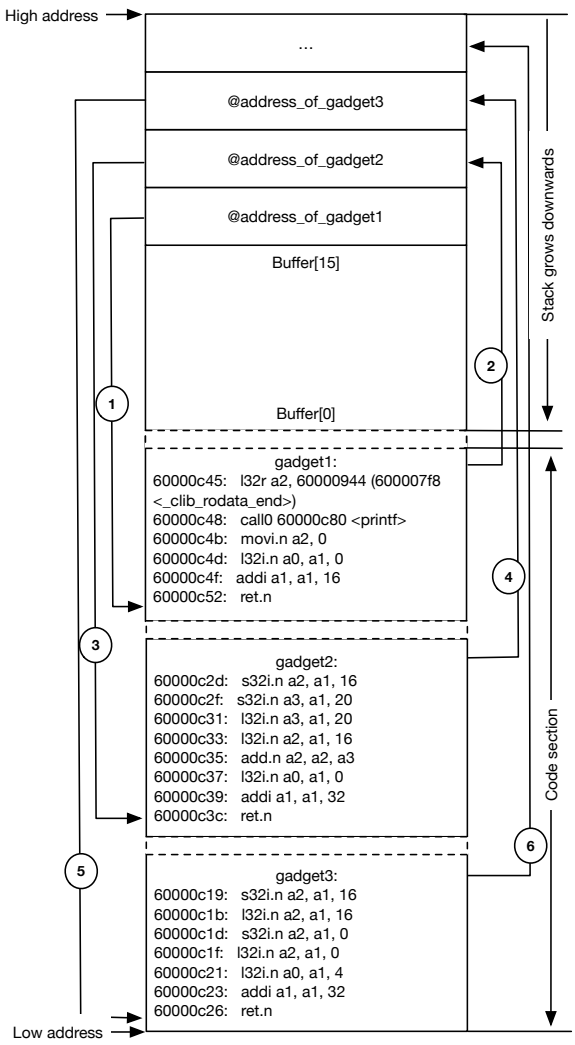


FIGURE 6.1: Xtensa ROP attack process.

flow is hijacked is labeled from *1* to *6*. In the end, the execution of these gadgets prints a string, adds the contents of two registers and initializes a register with a value from another register. If these gadgets were functions, it would be important not to use the starting address at the beginning of a function because this would result in infinitely returning to the same function – we do not want the `ret` to chain a gadget to itself repeatedly but to other meaningful gadgets. Essentially in Xtensa, the instructions occupying the first 5 bytes of every function reserve space for the function on the stack. This characteristic is the foundation of gadget chaining and simulating ROP behavior in Xtensa. Therefore, we always skip these instructions and addresses when crafting a payload for a ROP attack.

Besides, as shown in Figure 6.1, the actual gadgets normally reside in non-consecutive locations (as indicated by the dotted lines between them) in the code section of the memory, while the gadget's addresses could be in a consecutive location on the stack. Some garbage

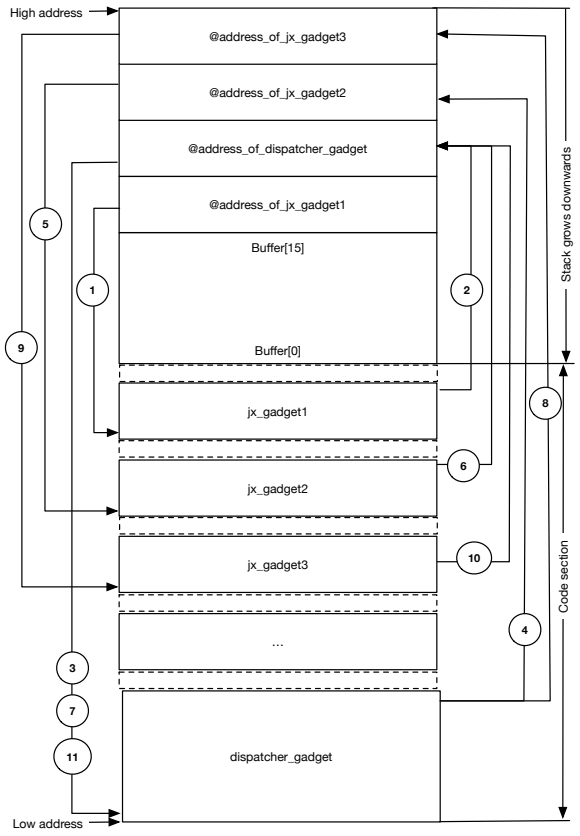


FIGURE 6.2: Xtensa JOP attack process.

addresses may be included as part of the actual gadget addresses on the stack which solely serve the purpose of address padding so that each gadget address is positioned at the top of the stack. Likewise, within the executable gadgets, some instructions exist as side effects, meaning they are not part of the intended exploit but they can also alter register states. Examples of such instructions were in the previous gadgets presented e.g line 2 of both Listing 6.3 and Listing 6.4 were unintended.

6.4.3 Xtensa JOP Attack Process

JOP uses gadgets ending with an indirect jump to an address in a register as demonstrated in Listing 6.3. However, the steps involved differ, as jump gadgets cannot be redirected to the stack with a return instruction, so to logically connect gadgets, we adopted the JOP model from [23]. That approach recommends that a *dispatcher gadget* is needed to link all jump gadgets, i.e., a *trampoline* gadget that relays from one jump gadget (also called *functional gadget*) to the next. In that scheme, the dispatcher gadget may maintain a *dispatch table*, which stores the JOP gadget addresses that should be executed sequentially, and each gadget must always point back to the dispatcher after execution. Figure 6.2 shows how these iterative steps can be implemented for JOP in Xtensa, and the label *1* to *8* represents the order of execution of instructions once JOP is initiated. Similar to ROP, a vulnerable

```
1 60001525:   addi.n a15, a15, 4
2 60001527:   add.n a1, a1, a15
3 60001529:   l32i.n a3, a1, 0
4 6000152b:   sub a1, a1, a15
5 6000152e:   jx a3
```

LISTING 6.5: A dispatcher gadget

```
1 60001555:   l32r a2, 600010ac
2 60001558:   call0 60001688 <printf>
3 6000155b:   jx a14
```

LISTING 6.6: A functional gadget

function not performing bounds checking can be used to launch the sequence, in that the return address of the vulnerable function is overwritten to point to the first jump gadget or the dispatcher, depending on the intent of the attack. An example of a dispatcher gadget in Xtensa is shown in Listing 6.5, this dispatcher gadget increases the value of `a15` – a regular register leveraged as an instruction pointer – by a constant (4), then points the stack pointer (`a1`) to the next address, loads the new address into `a3`, and jumps to this next functional gadget every time it is executed. This means that the dispatcher gadget can compute the addresses and jump to each of the functional gadgets `jx_gadget1`, `jx_gadget2`, ..., respectively.

An example of a gadget that could serve as a functional gadget is shown in Listing 6.6. The functional gadget loads the address of a string literal into `a2`, prints the string, and jumps back to the dispatcher gadget at `a14`. Unlike ROP, the addresses of JOP gadgets (other than the first) do not have to be placed on the stack, as they can be computed by the dispatcher gadget, in which case there is no need to retrieve the addresses of the executed instructions from the stack.

6.4.4 ROP Attack Detection Using HPC and Machine Learning

As explained in section 2.9 HPC can be used for vulnerability detection besides being used for debugging. Hence, in this section, we explain how HPC is used to detect the ROP and JOP attacks for the firmware-only embedded device programs on the Xtensa processor with the help of machine learning classier.

In Xtensa the performance monitoring library allows a total of 8 events to be monitored simultaneously and it can be accessed by using the `xt_perfmon` API. Xtensa LX7 has 30 main events with a total of 125 masks or sub-events, which represent the microarchitectural state triggered by any given running program. The available HPC events on Xtensa are shown in Table 6.1, and it should be noted that the labels and arrangement of the HPCs are random. Also, there are two prominent sources of noise generally associated with HPC readings. Those are related to the *program design* – noise caused by other internal or external

TABLE 6.1: List of available HPCs on Xtensa

Label	HPC: XTPERF_CNT_...	Interpretation
F1	COMMITTED_INSN	Instructions committed
F2	BRANCH_PENALTY	Branch penalty cycles
F3	MULTIPLE_LS	Multiple Load or Store
F4	INSN_LENGTH	Instruction length counters
F5	CYCLES	Count cycles
F6	PREFETCH	Prefetch events
F7	INSN	Successfully completed instructions
F8	PIPELINE_INTERLOCKS	Pipeline interlocks cycles
F9	D_ACCESS_U1	Data memory accesses (load, store, S32C1I, etc; load-store unit 1)
F10	D_ACCESS_U2	Data memory accesses (load, store, S32C1I, etc; load-store unit 2)
F11	D_ACCESS_U3	Data memory accesses (load, store, S32C1I, etc; load-store unit 3)
F12	D_STORE_U1	Data memory store instruction (load-store unit 1)
F13	D_STORE_U2	Data memory store instruction (load-store unit 2)
F14	D_STORE_U3	Data memory store instruction (load-store unit 3)
F15	D_LOAD_U1	Data memory load instruction (load-store unit 1)
F16	D_LOAD_U2	Data memory load instruction (load-store unit 2)
F17	ICACHE_MISSES	ICache misses penalty in cycles
F18	DCACHE_MISSES	DCache misses penalty in cycles
F19	OUTBOUND_PIF	Outbound PIF transactions
F20	OVERFLOW	Overflow of counter n-1 (assuming this is counter n)
F21	D_STALL	Data-related GlobalStall cycles
F22	I_STALL	Instruction-related and other GlobalStall cycles
F23	BUBBLES	Hold and other bubble cycles
F24	I_TLB	Instruction TLB Accesses (per instruction retiring)
F25	EXR	Exceptions and pipeline replays
F26	IDMA	iDMA counters
F27	D_TLB	Data TLB accesses
F28	I_MEM	Instruction memory accesses (per instruction retiring)
F29	INBOUND_PIF	Inbound PIF transactions
F30	D_LOAD_U3	Data memory load instruction (load-store unit 3)

instructions and programs – and the *HPC access* – noise caused by the reading of the HPCs.

The effect of these noises can be reduced by reading the right HPCs at the right place and correctly. Usually, not all of the instructions in a program under attack are malicious but only a fraction. Other instructions that are not affected by the attack are regarded as noise. To address this problem, we performed experiments by instrumenting code in which the approximate position of the start of the attack marks the beginning of the ROP or JOP, and the codes preceding and succeeding these executions are marked as benign as shown in Figure 6.3. In ROP, for instance, the benign section exhibits the actual behavior of the program. The behavior of the vulnerable section might be normal or malicious depending on whether it is exploited or not. *HPC recorder* records the microarchitectural events count. Targeting the HPC readings close to where the ROP execution is initiated allows us to have a more fine-grained and accurate HPC measurement which helps a classifier to correctly predict ROP or JOP behavior with high precision.

Hence, after collecting the data from the HPC measurement, we trained it using machine

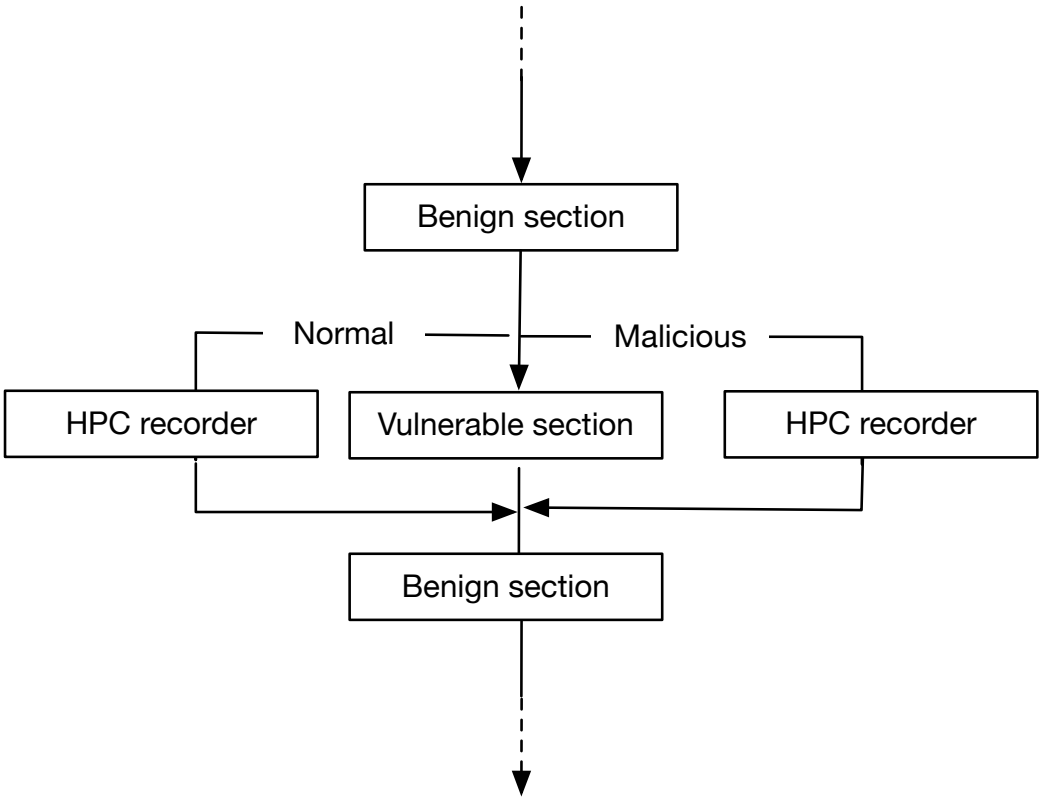


FIGURE 6.3: Instrumented code flow

learning (ML) to select an optimal model and detect the ROP and JOP attacks more accurately using the optimal model. We use 10-fold cross-validation for the training, i.e., 10% of the data is used for testing, which is a standard procedure to ensure the validity of the learned classifier. The detailed evaluation and comparison of the ROP attack detection using the machine learning technique is presented in section 6.5.3.

6.5 Evaluation

In this section, we discuss the research questions, the evaluation methodology, the experiment set-up, and the evaluation of the ROP and JOP attacks classifier. .

6.5.1 Research Questions and Evaluation Methodology

Research Questions: We present the research question as follows:

RQ1: What are the top HPCs in terms of indicating ROP and JOP behavior on Xtensa? We do not intend to use all HPC events even if we could, but only a minimal number that results in a good prediction. Using fewer HPC events means requiring fewer resources for detection.

RQ2: At what precision and recall can a classifier predict unusual behavior caused by ROP/JOP attacks within a running firmware code using the HPC events? In reality, the

HPC data for positive attacks scenarios is just a small fraction of the actual application data. In addition, we prefer detecting the malicious behavior as soon as possible (before the end of the program's execution), therefore, a classifier that can predict malicious behavior with a high true positive rate from the HPC events is desired.

Evaluation Methodology

Answering the first research question requires reading the right HPCs at the right place and correctly, i.e, we start reading the HPC events when the actual ROP and JOP attacks start by instrumenting the code to illustrate the start and end of the real attacks. A program would usually run multiple times as normal and malicious, and the machine learning classification method we use allows us to assign binary labels to the HPCs associated with the different executions. Also, we discovered that constant noise values were automatically added to our readings by accessing the HPC, these noise values were accounted for in both the benign and ROP execution. Our instrumented program contains different numbers of consecutive *push-pop* for ROP exploits and *jump* for JOP exploits.

To answer the second research question, the Xtensa ROP-JOP attacks classifier is evaluated using the following standard ML classification metrics – precision and recall. The metrics are defined as follows, where TP, FP, TN, FN refer to true positive, false positive, true negative, and false negative respectively.

Precision: This measures the accuracy of the positive predictions and for this, we want to know how many of the classified characteristics, recorded as HPC belong to the positive class. This metric is computed as:

$$Precision = \frac{TP}{TP + FP} \tag{6.1}$$

Recall: This is also known as true positive rate or sensitivity it is the ratio of positive instances that are correctly predicted as ROP or JOP by the classifier and it is computed as:

$$Recall = \frac{TP}{TP + FN} \tag{6.2}$$

6.5.2 Experimental Setup

The experimental setup consists of an Xtensa LX7 processor configuration designed with Xtensa Xplorer version 8.0.10.3000. This configuration runs on a Xilinx Zynq XCZ7020-1CLG484C[4] System on Chip (SoC) Module attached to a TE0703-06[5] carrier board. A Tincan

[4] https://wiki.trenz-electronic.de/display/PD/TE0720+Resources
[5] https://wiki.trenz-electronic.de/display/PD/TE0703+Resources

Flyswatter2[6] debugger, which provides an external Joint Test Action Group (JTAG) standard interface is connected to the carrier board for direct debugging of the programs. An SoC module is necessary because we experimented with the latest Xtensa processor generation, which was not available on the market at the time of carrying out this research. The embedded hardware configuration used is also minimal as not all of the features available in high-end IoT systems were available.

Programs and input

We train our model on breadth-first search (*BFS*) algorithm-based programs, in which we instrumented six versions of attack codes using varying ROP chains. This decision was based on observations from our several experiments that merging and training with data from different programs introduces unwanted statistical bias, which negatively affects model convergence. Therefore, in these six programs *P1*, . . . , *P6*, with a ROP chain length of 1, . . . , 6 are exploited, respectively. The programs were written in the C language, compiled using the Xtensa C compiler `xt-xcc`, and they run directly on the Xtensa processor. `xt-xcc` uses the GNU preprocessor, assembler, and linker but in addition, it provides a superior and smaller compiled code [26]. Malicious modification is made to the programs by the addition of an extra function call that simulates attacks such as buffer overflow and return-to-libc [133]. To flout the LIFO mechanism of the stack so that the program's control flow is hijacked, we corrupted the stack and modeled the payload to not just include gadget addresses to divert the control flow but also potential function arguments. Our design is mainly targeted at buffer overflow attacks to exploit memory corruption vulnerabilities because it is the most commonly used method, even in IoT devices [119, 46, 19]. The aftermath of the attacks then launches a ROP/JOP sequence that results in the execution of codes and functions in the programs that were originally never invoked. In Fig. 6.4, the execution time overhead in the training programs appears to increase when there are more frequent indirect calls to functions consisting of a few instructions. This instrumentation impact is similar to what was obtained for the ARM embedded benchmark used in [94].

To benchmark our model, we use a blend of 10 programs listed in Table 6.2, each running with a different set of inputs and payloads. Similar programs can be found in the CTuning suite[7] and MiBenchmark[8].The total size of the 10 programs is $\approx$ 2.74 MB. In Table 6.2, *ET1*, *ET2*, *Ovd*, *#Rop*, and $\mathcal{O}$ *notation* represents the original execution time, execution time with HPC measurement, instrumentation overhead, the length of the rop/jop chain, and the time complexity of the programs, respectively. The maximum and minimum overhead recorded of 1.34% and 0.74%, respectively, look reasonable. We do not directly compare

[6] `https://www.tincantools.com/product/flyswatter2`
[7] `http://ctuning.org/`
[8] `https://vhosts.eecs.umich.edu/mibench/`

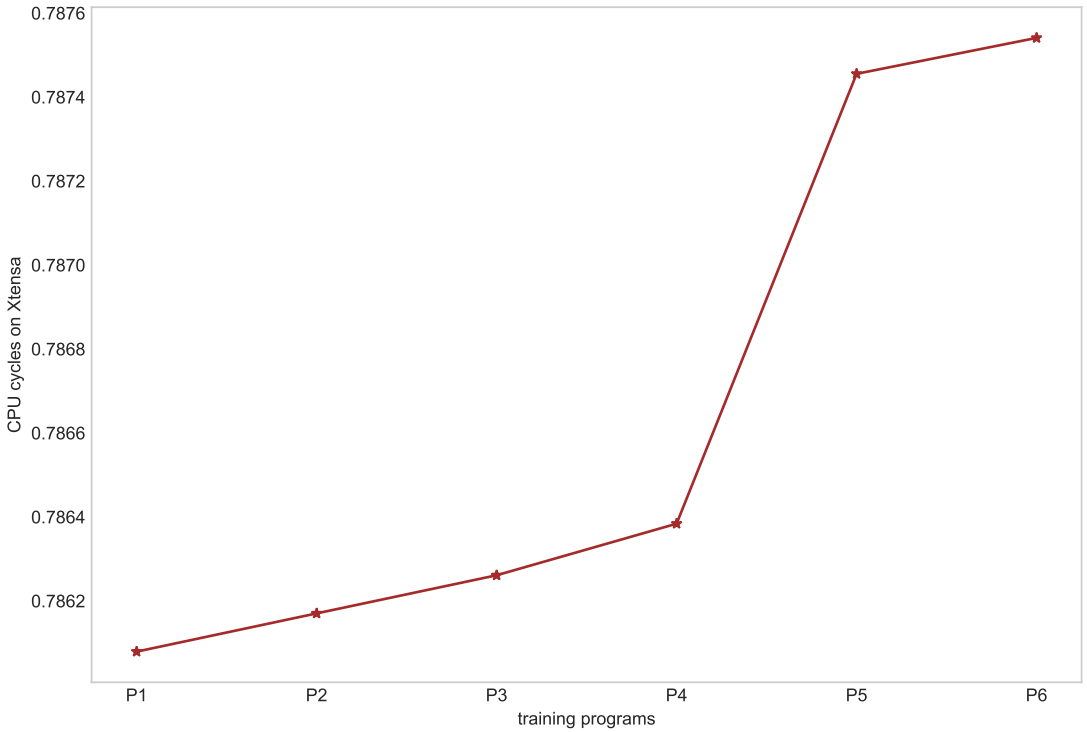


FIGURE 6.4: Training programs running time on Xtensa

these programs' performance because they differ in input and runtime complexities, our main goal is to see how the ML model will perform against random programs like these.

6.5.3 Data and model selection:

To find the optimal model for the attack detection, we first recorded the events that were triggered during multiple executions of the exploited programs for ROP and JOP attacks, respectively. The HPC data was recorded at 5 frequencies ($\approx 10 * 2^{12}$ to $50 * 2^{12}$), which allows us to sample data from a wide corpus of cycles, and thus provides a sufficient dataset that best represents the behavior of the programs under attack. On Xtensa, the frequency parameter is expected to be a multiple of 2^{12} to prevent round-off errors. We started at $\approx 10 * 2^{12}$ because this frequency has been used to validate the integrity of program control flow via HPC with some promising results [91]. For any given program, the higher the frequency chosen, the lower the noise effect, as well as the number of HPC sample size recorded. Our data shape is (6061, 30), containing 6061 rows of event counts and 30 features. The support vector ML (SVM) algorithm is preferred because it excels for data in high dimension spaces and it is relatively memory efficient. SVM is an excellent binary classifier if data is balanced but because the positive cases are less than the negative cases, with about a factor of 7, we use a weighted SVM. The weighted SVM modifes the SVM penalty parameter C_i to fit the model for each instance i, so that the weight w_i is proportional to the class distribution. We use 10-fold cross-validation, which is a standard procedure to ensure the validity of the

TABLE 6.2: Benchmark programs

Program	ET1 (μs)	ET22(μs)	Ovd(%)	#Rop	$\mathcal{O}$ notation
DFS	4421664	4481007	1.34	2	$\mathcal{O}(V+E)$
Kruskal	2273937	2297769	1.05	3	$\mathcal{O}(E \log E)$
RabinKarp	642475	647202	0.74	4	$\mathcal{O}(mn)$
Huffman	2343248	2368711	1.09	5	$\mathcal{O}(n \log n)$
Mergesort	933301	941449	0.87	6	$\mathcal{O}(n \log n)$
LCS	677565	688501	1.61	1	$\mathcal{O}(mn)$
Prim	1577938	1596889	1.20	2	$\mathcal{O}(E \log V)$
BinaryS	507317	511553	0.83	4	$\mathcal{O}(\log n)$
FloydWarshall	1748544	1765759	0.93	5	$\mathcal{O}(n^3)$
BellmanFord	1448642	1460045	0.79	6	$\mathcal{O}(VE)$

learned classifier. The model performance measured by the mean of the ROCAUC score is 0.94, which is well above 0.5, this means the classifier has a predictive ability.

We conducted the ML experiments on a MacBook Pro with a 2.9 GHz Intel Core i7 processor and 16GB RAM.

6.5.4 Discussion

In this section, we discuss our findings and their contribution to the research questions.

Important HPC events: Of the 30 main HPC events in Table 6.1, feature engineering found that the readings for events F1, F3, F4, and F7 are the same values irrespective of the number of times a program executes either as benign or attacked code. An explanation for the similarities in the HPC values could be that the low-level events (masks) in these main events being accounted for occurred at almost the same count rate. Therefore, F1, F3, F4, and F7 serve as the pivot for the permutation with the remaining events, together with all the sub-events, to determine which events are dependent on them. Notwithstanding, the HPC values recorded are reproducible for any given program, the reasons for the deterministic nature of some of the counted and recorded events could be the result of (a) running the programs on a bare board with no operating system or kernel and (b) there are no running background services, which reduces the effect of interference in the readings.

The distribution of the 8 candidate HPC events (used in our final SVM model) in the benign and attack code is represented in Fig. 6.5 by violin plots, which are combinations of box-and-whisker plots and probability density functions (PDF). The violin plot shows the density and distribution of the readings for each of the 8 selected HPC events. -1 and 1 represent HPC events in the benign and attack executions respectively. Fig. 6.5a, Fig. 6.5b, Fig. 6.5c, Fig. 6.5d, Fig. 6.5e, Fig. 6.5f, Fig. 6.5g and Fig. 6.5h are the distributions for F1, F2, F5, F8, F12, F15, F25 and F27 respectively. While at a glance some features such as F2, F12, F15, and F27 might look close for both the benign and ROP runs, dropping them degrades the

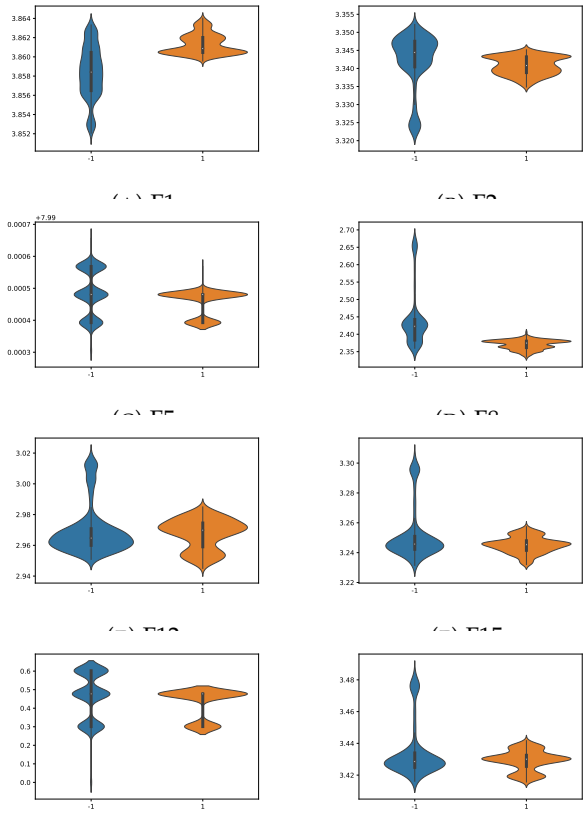
model's performance (the overall recall falls from 70% to 58%.) However, it may of course be possible, in the future, to use fewer than 8 HPC events, but our interest at the moment is to identify the best events that Xtensa's 8 performance counter registers could monitor simultaneously, and which could give a strong indication of ROP/JOP execution on an embedded system running a firmware. In the sequel we give an overview of the selected HPC events:

- **F1** in Xtensa is equivalent to the number of retired instructions and it is the number of instructions reaching the *W* stage without being killed at a given sampling interval.
- **F2** relates to the number of branch penalty instruction events in a given sampling interval.
- **F5** distribution appears to cover two and three PDF regions in the benign and ROP-affected run respectively.
- **F8** relates to the number of stalls in the pipeline in a sampling interval.
- **F12** records the number of stored instructions events (such as store misses and cached store) from the data memory in a sampling interval.
- **F15** records the number of the load instruction events (such as load misses and cache load) from the data memory in a sampling interval.
- **F25** can record, for example, the number of the exceptions, interrupts, and replays resulting from TBL misses, load and store errors, illegal instructions, etc.
- **F27** records the number of lookups to the data translation lookaside buffer (TLB)

More elaborated explanation of the selected HPC events is found in Appendix A.1

Model performance metrics: Our classifier is trained to predict if illegal instructions using *returns* and *jumps* have been exploited and this is put to test against some benchmark programs. The model is fed with HPC data of these programs running on the bare-metal Xtensa (on the FPGA).

Fig. 6.6a and Fig. 6.6b show the individual precision and recall of the classifier. In Fig. 6.6a the precision peaked for *floydwarshall, and bellmanford* where a higher number of illegal instructions was executed. While it is intuitive that the metrics improve with the increase in *returns/jumps*, we found this not to be always true if the exploited programs are different. That is why *BinaryS* with 4 rop/jop violations has lower precision (0.85) than *Prim* (1.00) and LCS (0.94) with 1 and 2 violations, respectively. We verified this and the metrics actually increase with more *returns/jumps* exploitation in the same program. *DFS* has the lowest precision of 0.32, however, this is not a problem as long as the recall is high, this is based on the premise that in reality, the positive case is rare, and hence the percentage of the positive instances that are correctly detected should be optimized. The recall for DFS is almost twice as high, 0.60. The top three recall values are for *huffman* (0.84), *prim* (0.83)



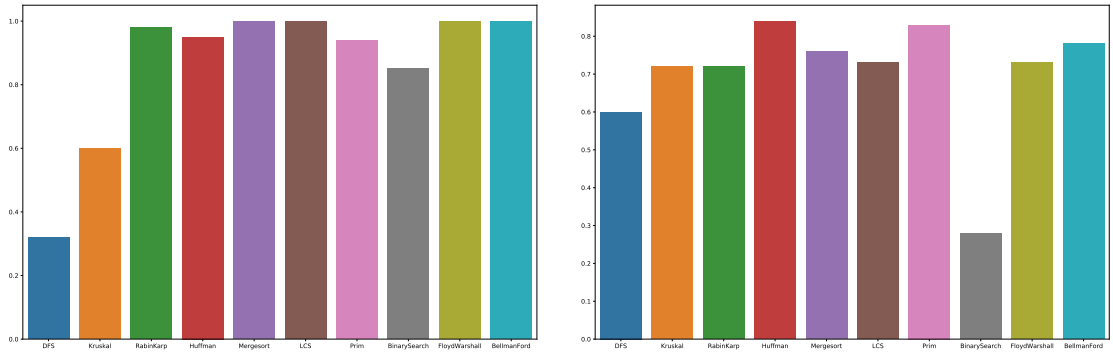
(G) F25 (H) F27

FIGURE 6.5: Box-and-whisker plots of the event counts for the benign (-1) and ROP executions (1)

and *bellmanford* (0.78), these are very interesting because it means that almost all of the ROP instances are correctly detected. However, *BinaryS* has a low recall of about 0.30. We examined this and it appears the *BinaryS*, which is already fast with $\mathcal{O}(n \log n)$ running time, also ran on a small input array of length 12. From Table 6.2 it appears that, even with the instrumentation overhead, it still runs fast. It appears that the complexity of the program did affect the capability of the classifier. However the overall detection average accuracy of 0.79 is significant.

Threats to Validity We identify the following potential threats to validity:

- Examples of ROP/JOP exploit code for non-standard processors are not easy to find in practice, and therefore the instrumented programs used for our training may not be representative of all ROP programs' behaviors on Xtensa. Notwithstanding, the size of our data together with the results obtained could reasonably justify the reliability of our model in detecting attacks initiated through a buffer overflow, which is the most exploited vulnerability.



(A) Precision (B) Recall

FIGURE 6.6: Evaluation on benchmark programs

- Our ROP/JOP exploits do not consider specific attack cases such as accessing a shell, which might be the final intention of the attacker even though it may not be available in the context of embedded/IoT devices, which often only contain firmware.

Chapter 7

Related Work

JavaScript static analysis frameworks. TAJS [70], JSAI [74], WALA [121, 142, 48, 128], and SAFE [82, 116, 114] are JavaScript static analysis frameworks among which WALA and SAFE are most commonly used in research projects according to citation numbers. WALA provides soundy flow-insensitive static analysis supporting correlation tracking [142] to improve the analysis's scalability and precision. Additionally, WALA introduced a new unsound but more scalable static analysis that performs field-based (FB) call graph [48]. SAFE [116] is a flow- and context-sensitive scalable static analysis framework which supports loop-sensitivity analysis (LSA) [114]. LSA enhances the analysis precision in loops improving analysis scalability of JavaScript applications. The current version SAFE 2.0 [116] supports *pluggability*, *extensibility* and *debuggability* which makes it more user-friendly than WALA and SAFE 1.0. SAFE leverages a *recency abstraction* [16, 115], which performs strong updates on recently allocated objects and weak updates on joined old objects.

Points-to Analysis Comparison. The average points-to set size analysis is a convenient way to evaluate the precision of static analysis frameworks. For instance, Wei et. al. [167] used the average points-to comparison to systematically select a precise context-sensitivity type (call-site, object, and parameter) per function. A context-sensitivity type that provides the smallest average points-to set size is selected for each function. This motivated us to evaluate the precision and scalability trade-off between WALA and SAFE using average points-to set sizes. However, both frameworks follow different points-to analysis approaches and leverage different data structure representations, which makes the comparison non-trivial. Hence, we use the approach followed by [80] to integrate the WALA analysis results into SAFE. This simplifies the selection of comparable objects. [80] also evaluates the precision of WALA's FB, SAFE and their combinations based on the average number of callees of the call sites. In contrast, we use the object properties and their points-to set to evaluate the precision of WALA's PB and SAFE. The object properties include the callees and other fields which broadens our analysis scope. Moreover, unlike the related work, which integrates

WALA with SAFE 1.0, we integrate WALA with the latest, written completely from scratch, version SAFE 2.0 to identify the comparable objects in both analyzers.

Static Taint analysis. SAFE's existing taint analysis [123] forms the basis of our approach. However, it does not support primitive arguments reaching the sinks, nor sanitizers. In contrast, our analysis supports object arguments. Additionally, we modeled *JSON.stringify*, and our analysis correctly identifies when the tainted user input is sanitized using this function. Taint analysis is well-established for security purposes. In the domain of JavaScript, however, due to its dynamic nature, there is only limited related work [8] leveraging static analysis: [73] and [127] present dynamic analyses to detect illicit taint flows, which can only assert security for the given program execution. [166] leverage a combination of static and dynamic analysis to identify security vulnerabilities due to data integrity violations in JavaScript codes in websites. Again, the dynamic analysis component may miss certain security-relevant facts. Skoruppa et al. [15] detect security violations in an online voting client written in JavaScript via pure static analysis with WALA. None of these approaches, though, analyzes module-based code based on node.js, which increases complexity significantly. Besides, we are the first to assess the precision and scalability of different static analysis frameworks precisely. This allows us to scale the analysis to even more complex language features, which we are actively pursuing at the time of this writing.

Dynamic Taint Tracking. Dynamic taint tracking has been proposed as a runtime approach to detect code injection attacks in production applications, as a sort of last line of defense [138, 55, 147, 127, 17, 93]. However, these approaches are generally not adopted due to prohibitive runtime overhead: even the most performant can impose a slowdown of at least 10–20% and often far more [32, 17, 76, 45]. Although prior work has used the term test *amplification* to refer to techniques that automatically inject exceptions or system callbacks in existing tests [176, 175, 2], we believe that RIVULET is the first to use dynamic taint tracking to amplify test cases.

Testing for Injection Vulnerabilities. A variety of automated testing tools have been proposed to detect injection vulnerabilities before software is deployed. These tools differ from black-box testing tools in that they assume that the tester has access to the application server, allowing the tool to gather more precise feedback about the success of any given attack. Kieżun *et al.*'s Ardilla detects SQL injection and XSS vulnerabilities in PHP-based web applications through a white-box testing approach [77]. Ardilla uses symbolic execution to explore different application states, then for each state, uses dynamic taint tracking to identify which user-controlled inputs flow to sensitive sinks, generating attack payloads for those inputs from a dictionary of over 100 attack strings. Similar to Ardilla, Alhuzali *et al.*'s *Navex* automatically detects injection vulnerabilities in PHP code using concolic execution to generate sequences of HTTP requests that reach vulnerable sinks [4]. RIVULET

improves on these approaches by leveraging the context of the complete value flowing into each vulnerable sink, allowing it to focus its payload generation to exclude infeasible attack strings. The naive rerun generator that we used as a comparison in our experiments roughly represents the number of attack strings that Ardilla would have tested, showing that RIVULET provides a significant reduction inputs tested. Unlike these systems' automated input generators, RIVULET uses developer-provided functional tests to perform its initial exploration of the application's behavior, a technique that we found to work quite well. If a more robust concolic execution tool were available for Java, it would be quite interesting to apply a similar approach to RIVULET, which could reduce our reliance on developer-provided test cases to discover application behavior.

Other tools treat the application under test as a black-box, testing for vulnerabilities by generating inputs and observing commands as they are sent to SQL servers, or HTML as it is returned to browsers. Mohammadi *et al.* used a grammar-based approach to generate over 200 XSS attack strings, however, our context-sensitive approach considers the location of taint tags within the resulting document, allowing RIVULET to select far fewer payloads for testing [96]. Simos *et al.* combined a grammar-based approach for generating SQL injection attack strings with a combinatorial testing methodology for testing applications for SQL injection vulnerabilities [135]. Thomé *et al.*'s evolutionary fuzzer generates inputs to trigger SQL injection vulnerabilities using a web crawler [155]. Others have considered mutation-based approaches to detect SQL injection [12] and XML injection vulnerabilities [68]. In contrast, RIVULET uses data flow information to target only inputs that flow to vulnerable sinks.

While our work considers injection vulnerabilities that are triggered through code that runs on a web server, other work focuses on injection vulnerabilities that exist entirely in code that runs in client browsers. Lekies *et al.* deployed a taint tracking engine inside of a web browser, traced which data sources could flow into vulnerable sinks, and then generated XSS attacks based on the HTML and JavaScript context surrounding each value at the sink [87]. RIVULET also uses taint tracking to generate attack payloads, expanding this approach to generate SQL and RCE injection attacks, and uses existing test cases to expose non-trivial application behavior.

A variety of static taint analysis approaches have also been used to detect injection vulnerabilities [139, 158, 141, 13]. The most recent and relevant is *Julia*, which uses an interprocedural, flow-sensitive and context-sensitive static analysis to detect injection vulnerabilities [139]. Compared to a dynamic approach like RIVULET, static approaches have the advantage of not needing to execute the code under analysis. However, in the presence of reflection, deep class hierarchies, and dynamic code generation (all of which are often present in large

Java web applications), static tools tend to struggle to balance between false positives and false negatives. In our benchmark evaluation, we found that RIVULET outperformed Julia.

While RIVULET uses specialized input generation and attack detection to find code injection vulnerabilities, a variety of fuzzers use taint tracking to instead find program crashes. For instance, BuzzFuzz uses taint tracking to target input bytes that flow to a sink and replace those bytes with large, small, and zero-valued integers [49]. VUzzer takes a similar approach, but records values that inputs are compared to in branches and uses those same values as inputs (*e.g.*, if it sees `if(taintedData[49] == 105)...` it would try value 105 in taintedData byte 49) [120]. Similarly, TaintScope uses fuzzing to detect cases where fuzzed inputs flow through checksum-like routines and uses a symbolic representation of these checksum bytes when generating new inputs in order to pass input validation [162]. RIVULET's key novelties over existing taint-based fuzzers are its context-sensitive input generation which enables the creation of complex, relevant attacks and its attack detectors which report injection vulnerabilities rather than just program crashes.

Gadgets discovering and chaining. Several tools have been developed for gadgets discovering and chaining to build ROP attacks on various architectures. `ROPgadget`[1], `Ropper`[2] and `xrop`[3] are some of the tools that are used for gadget discovery and are still in active development. Each of them supports the *ELF/PE/Mach-O* format on the x86, x64, ARM, PowerPC, SPARC, and MIPS architectures. In addition to gadget discovery, both ROPGadget and Ropper support chaining gadgets on x86 and x64 to build ROP exploits automatically. To the best of our knowledge, `xrop` is the only available automated tool for generating gadgets on the Xtensa processor, but it does have it's limitations and it is not supported by the popular Capstone-Engine[4] which is the multi-architecture and multi-platform disassembler used by many of the publicly available gadget tools. Our gadgets compilation in Xtensa platform is based on `xrop` with additional gadgets discovery options as discussed in Section 6.4.1.

ROP Attack Detection on x86.

Recently, several hardware-based ROP defense tools such as HDROP [178], SIGDROP [163], HadROP [117], ROPSentry [37] were proposed for x86. They use heuristics or machine learning models which leverage branch misprediction events that occur at return instructions. HDROP utilizes HPCs such as mispredicted return events to defend against ROP exploits. However, it requires the instrumentation of source code to insert checkpoints and provides substantially high overhead. Later on, SIGDROP was proposed, which has strict policies to leverage HPC to efficiently capture and extract the signatures to detect ROP

[1] https://github.com/JonathanSalwan/ROPgadget
[3] https://github.com/jsandin/xrop
[4] https://www.capstone-engine.org

attacks. However, the policies can be bypassed by a determined adversary, for example, by inserting one redundant call-ret paired gadget without causing any misprediction at the return instruction.

The most closely related solution to our work, HadROP [117] was proposed by Pfaff et al., which uses machine learning techniques to generate a kernel module that detects and prevents ROP attacks at runtime using HPC as input data. However, our solution focuses on the more challenging ARM architecture, which is leading the market and capable of outperforming x86 [53]. We also feed much larger data sets from real-world applications into the offline learning technique than HadROP. Unlike HadROP, we also evaluated our SVM model with respect to seven other machine learning techniques.

Das et al. proposed ROPSentry [37], a defense framework, which can detect ROP by analyzing the ROP exploits and spraying attacks using hardware events and reduces the performance overhead by an adaptive and a return miss-based sampling technique, i.e., fetching HPC values at every return miss. But similar to HadROP, ROPSentry is only available for x86, not for ARM.

ROP attack detection on ARM. As explained earlier, most ROP detection approaches are based on x86. However, there are also initial results on the ARM platform. The work of Huage et. al. [65] is one of the earlier ROP attack detection approaches using dynamic binary instrumentation (DBI), which, however, induces a high performance overhead. To overcome this limitation Lee et. al. [83] proposed a *meta-data* driven approach that uses the ARM CoreSight traces supplemented with offline binary analysis to generate meta-data information missed in the debug traces. Then using the information from the meta-data and the debug traces they apply the shadow call stack (SCS) [111] approach to verify the integrity of the direct and indirect call/jump instructions and detect ROP attacks. However, since this needs high memory/storage overhead they tried to improve it in their later papers [84, 85] by instrumenting the binary in the way CoreSight debugger traces provide full information required for the control flow verification. These papers support ROP attack detection using the SCS [111] approach and JOP attack detection using the Branch Regulation (BR) [75] approach. More recently CFVerifier [102] was proposed to overcome the storage overhead caused by the meta-data in [83] by maintaining table entries only for branch instructions instead of every instruction.

Unfortunately, it has been shown [41] that these detection approaches can be circumvented via advanced attacks such as print-oriented programming [29] attacks, counterfeit object-oriented programming [129], or data-oriented programming [63], which are not directly related to the branch integrity. Besides, most of the approaches use the ARM CoreSight debugger based hardware monitor which could drop traces given a sufficiently high branch rate since the monitor requires more time to process a trace than the rate at which branches

occur on the target processor [41]. Another limitation of using debugger traces to detect CRA attacks is that the hardware debugger can be used by an attacker to circumvent the security of the system. If the attacker can access the debug interface, he could use it to tamper with the code and data memory, or even disable the hardware monitor by tampering with the tracing mechanism [41]. Moreover, most of these papers use Branch Regulation (BR) analysis, which provides only partial indirect branch protection since indirect branches to any address within the current function are allowed. This leaves BR somewhat vulnerable to unintended branches since it allows CRAs which do not cross function boundaries. In contrast, the HPC and machine learning-based approach does not use an external hardware debugger and cannot drop traces during the monitoring. Besides, it does not use any meta-data that leads to storage overhead. Moreover, since it is using machine learning it is not specific to a set of given attack types or branch regulations. To the best of our knowledge we are the first to investigate ROP attack detection via HPCs and machine learning on the ARM platform.

ROP on Xtensa. To the best of our knowledge, only one article by Kai et. al. [86] proposes a basic ROP attack on the Xtensa processor by chaining simple functions considered as gadgets. In contrast, our thesis implements ROP and JOP attacks using compiled gadgets on the Xtensa LX7 architecture. Moreover, we are the first to design JOP attacks using a dispatcher gadget compatible with the Xtensa processor. Kai et. al. [86] showed that Xtensa can be attacked by chained gadgets irrespective of the ABI in use. The authors additionally proposed a linked list approach to chain gadgets for Xtensa's Call0 ABI. Since the approaches to attacking either ABI are similar, we used the default ABI in our thesis to demonstrate ROP/JOP on Xtensa. Their paper is also different in that it did not cover attack detection. Although based on a different method, a similar platform to ours is used in [94] that proposes a solution called FPGA CFI for bare-metal ARM embedded devices. Their approach targets devices that read firmware instructions directly from the flash memory but unlike our work, ROP detection was not included in their work. Additionally, CFI solutions' memory requirements and overheads are generally considered impractical to secure resource-constrained embedded devices [163].

Chapter 8

Conclusion

Despite many efforts to reduce their incidence in practice, code injection attack remains common, and was ranked #1 until 2021 on OWASP's most recent list of critical web application vulnerabilities [105]. Even after 2021, it is ranked in the top #3. Moreover, due to the fast growth of mobile technology and internet-of-things (IoT) [125] injection attacks such as ROP and its sibling JOP are also getting high attention in modern ARM and Xtensa Processors. Especially, the ARM's support of low power consumption without sacrificing performance is leading industries to shift towards ARM processors [53], which advances the attention of the attacks too.

In this thesis, we present different static and dynamic analysis approaches to provide precise code injection attack detection and prevention solutions for the attacks on real-world web applications and embedded applications based on ARM and Xtensa processors.

- **Static Security Evaluation of an Industrial Web Application**: In this work, we analyzed the security of a real-world hybrid app from our partner company by selecting a more precise static analysis framework and extending its taint analysis approach. To select the best analysis framework, we performed a thorough comparison of SAFE and WALA by integrating the analysis result of WALA into SAFE and computing the average points-to-set of pointers (object properties) considering user-defined objects, global variables and other comparable elements only. The average points-to-set analysis result indicates SAFE is very precise at the cost of some scalability. SAFE also provides better model coverage. For static analyses, precision is often more crucial than scalability (as long as it finishes execution in a given timeframe), especially for security-related analysis. Hence, based on the result of our comparison analysis, we selected SAFE to analyze the real-world hybrid app. Finally, to analyze the security of the hybrid app, we took some of its program slices with independent sources and performed taint analysis by simulating tainted values as @StrTop in SAFE. Since the existing taint analysis in SAFE supports only primitive values, we extend it to

identify tainted objects as well. As the result, our extended taint analysis tool correctly identified the flow of tainted values from sources to sinks in the hybrid app.

- **Revealing Injection Vulnerabilities by Leveraging Existing Tests:** We have presented a new approach to automatically detect injection attack vulnerabilities on web applications before the software is released by amplifying existing application tests with dynamic taint tracking. Rivulet applies novel, context-sensitive, input generators to efficiently and effectively test applications for injection vulnerabilities. On four benchmark suites, Rivulet had nearly perfect precision and recall, detecting every true vulnerability (except for one pathological case) and raising no false alarms. Using developer-provided integration tests, Rivulet found six new vulnerabilities and confirmed one old vulnerability in three large open-source applications. Rivulet is publicly available under the MIT license [62], and an artifact containing RIVULET and the experiments described in this thesis is also publicly available [61].
- **Detecting and Preventing ROP Attacks using Machine Learning on ARM:** This addresses the practicability of detecting and preventing ROP attacks using HPCs and machine learning on the ARM processor. First, we crafted several real-life exploits using ROP attacks from selected vulnerable programs, which provided HPC data about the execution behavior of these vulnerable programs. For the ROP attack executions, a small debugger tool called "tracer" was implemented to record only the real ROP attack execution part, i.e., after the first ROP gadget starts execution. The SVM RBF kernel is used for offline training and online monitoring of our ROP attack detection approach, as when the detection accuracy of the offline training is evaluated with respect to 7 additional machine learning techniques, it is consistently in the very top, i.e., it provides 92% detection accuracy and no one provides more than that. The detection accuracy and performance overhead of the online monitoring is also evaluated and it provides around 75% detection accuracy with an average 6.2% slowdown overhead. Last but not least, our ROP attack detection and prevention approach using HPC and machine learning techniques demonstrates that the characteristics of the hardware events on ARM processors can be used to investigate whether or not an attack is in progress.
- **Detecting ROP on Firmware-Only Embedded Device Using HPCs:** We have been able to demonstrate the possibilities of how the Xtensa Call0 ABI processor configuration could be attacked using gadgets from an executable linkable format binary of user programs. We extracted valid gadgets, demonstrated gadget chaining scenarios for return- and jump-oriented programming, and carried out experiments with these attack scenarios on a minimal Xtensa hardware configuration running as a bare-metal embedded system. Our hardware configuration is minimal and targeted for low configuration embedded/IoT devices running instructions from the flash memory. Furthermore, we experimented with the available hardware performance counter

events and trained a support vector machine classifier based on these HPCs to detect ROP and JOP readings in our test programs. By evaluating the model on unseen HPC data, we obtained high precision and recall. We also identified some HPCs which could help in predicting the execution of these kinds of code reuse attacks. Our validation results prove the feasibility of the SVM and HPC detection methods for ROP or JOP on Xtensa from a functional perspective, thereby validating the capacity of this technique to detect code reuse behavior on a firmware-only Xtensa processor.

Chapter 9

Future Work

In this section, we explain the potential extensions points

9.1 Extend the Scope of the Static Analysis tools Comparison

In this thesis, we have only compared WALA and SAFE to select a static analysis framework for JavaScript that precisely analyze the security of a real-world hybrid application since most of the scientific literature we had reviewed use these frameworks. However, it would be nice extending the scope of the framework evaluation to include other analysis frameworks such as TAJS to make the comparison more robust.

9.2 RIVELUT with Automatic Test Generation and Implicit Flow

Our tool RIVELUT is capable of detecting vulnerabilities from source-sink flows that are exposed by a test case. Hence, Rivulet requires applications to have existing test cases. Although we believe that this is a fair assumption to make, and in our evaluation, show that Rivulet can detect a real vulnerability even when presented with a very small test suite (for Apache Struts). Our approach can be extended with an automatic test generation technique to mitigate this limitation. Moreover, at present, RIVULET can only detect vulnerabilities that result from explicit (data) flow, and not through implicit (control) flows. This limitation can be mitigated in the future by supporting implicit flow tracking in Phosphor, which is used as a base taint analysis framework for RIVELUT.

9.3 Increase the Training Data set for ROP Attack Detection

Even though our ROP attack detection approach on ARM using HPC and machine learning has provided good detection accuracy in the offline training, the online detection accuracy has somehow decreased, potentially due to the fixed-point arithmetics we resorted to [21], and the size of training data. The machine learning classifier was trained based on data collected from 10 exploits from 5 real-world vulnerable applications and this might not be representative of all possible exploits. Hence to detect the ROP attack during the online monitoring more precisely, our approach can be extended by collecting more training data so that a more appropriate model is generated.

Similarly, for training our ROP/JOP attacks detection on Xtensa we collected the data from small instrumented programs which may not be representative of all ROP programs' behaviors on Xtensa. For future work, we recommend training the machine learning using data collected from more realistic ROP/JOP attacks.

Appendix A

Appendix

A.1 HPC Events Selected in Xtensa

F1 in Xtensa is equivalent to the number of retired instructions and it is the number of instructions reaching the *W* stage without being killed at a given sampling interval. At the *W* stage, the effect of an instruction on the architectural state is irreversible. The wider PDF region of the ROP infected run, which is at the same time above the third quartile of the benign run, is abnormal. The median of the ROP run also lies above the boxplot of the benign run, meaning that the two HPC data belong to a different group. The ROP run HPC values occur frequently in one PDF region and this is as a result of the execution of small gadgets performing little tasks and leading to slightly more/faster-committed instructions per interval.

F2 relates to the number of branch penalty instruction events in a given sampling interval. The pipeline will be stalled if more branch instructions are executed than they are taken. All of the branch instructions in the benign programs were correctly executed while the malicious program executes only a few selected branch instructions to accomplish its malicious intention. The boxplots look like they are slightly in the same range and symmetrical but the PDF region of the ROP-affected run shows the HPC values occur more frequently and this can be attributed to the more mispredicted branches.

F5 distribution appears to cover two and three PDF regions in the benign and ROP-affected run respectively. For the ROP-affected run, the median is almost identical to the third quartile which is why they overlap, this is likely because of a large proportion of low values of *F5* events. The ROP-run skipped some instructions and this could be responsible for the very low variance in this HPC.

F8 relates to the number of stalls in the pipeline in a sampling interval. In Xtensa, the number of interlocks refers to the number of R-stage holds arising from register dependencies and interlock-specific instructions. Register dependencies are low because gadgets skip several normal instructions. The median of the ROP-affected run is also unsurprisingly outside the

box of the benign run and the boxes' variance shows that this pipeline delay varies more in the benign run than in the ROP-affected run.

F12 records the number of stored instructions events (such as store misses and cached store) from the data memory in a sampling interval. The ROP run has significantly more variance and inverted PDF and this is likely caused by a reference to data not in the data memory.

F15 records the number of the load instruction events (such as load misses and cache load) from the data memory in a sampling interval. The data memory, unlike the instruction memory, is both readable and writeable. The normal run distribution have slightly more variance than the ROP-affected run, however, the PDF is an indication that the ROP-run performs data manipulation operations and load data operation more frequently.

F25 can record, for example, the number of the exceptions, interrupts, and replays resulting from TBL misses, load and store errors, illegal instructions, etc. It is not surprising that despite the median being the same for the two boxes, the ROP-affected run is severely skewed and the unusual PDF width shows that the majority of the *F25* HPC readings are in one region. This implies that the error handling by this HPC occurs more frequently in the ROP-affected run.

F27 records the number of lookups to the data translation lookaside buffer (TLB). Unlike the von-Neumann architecture, the Harvard architecture can have separate memory access hardware - instruction TLB and data TLB. Data TLB hit helps to reduce the data access time from the data memory. This HPC for the two runs appears to be in the same group but the PDF is irregular with a high-frequency region for the ROP run, this is because each data TLB miss leads to a computationally expensive page table lookup for the physical addresses of data.

Bibliography

[1] W $\wedge$ x ("write xor execute").

[2] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[3] Manaar Alam, Sayan Sinha, Sarani Bhattacharya, Swastika Dutta, Debdeep Mukhopadhyay, and Anupam Chattopadhyay. Rapper: Ransomware prevention via performance counters. *arXiv preprint arXiv:2004.01712*, 2020.

[4] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V. N. Venkatakrishnan. Navex: Precise and scalable exploit generation for dynamic web applications. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 377–392, Berkeley, CA, USA, 2018. USENIX Association.

[5] Tim Ambler and Nicholas Cloud. Browserify. In *JavaScript Frameworks for Modern Web Dev*, pages 101–120. Springer, 2015.

[6] Mahmoud Ammar, Giovanni Russello, and Bruno Crispo. Internet of things: A survey on the security of iot frameworks. *Journal of Information Security and Applications*, 38:8–27, 2018.

[7] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[8] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Computing Surveys (CSUR)*, 50(5):1–36, 2017.

[9] Apache Foundation. Apache struts. `https://struts.apache.org`, 2019.

[10] Apache Foundation. Apache struts release history. `https://struts.apache.org/releases.html`, 2019.

[11] Apache Foundation. Apache tomcat. `https://tomcat.apache.org`, 2019.

[12] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 259–269, New York, NY, USA, 2014. ACM.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[14] Itzhak Zuk Avraham. Non-executable stack arm exploitation research paper. *Revision*, 1:2010–2011, 2010.

[15] Michael Backes, Christian Hammer, David Pfaff, and Malte Skoruppa. Implementation-level analysis of the javascript helios voting client. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 2071–2078, 2016.

[16] Gogul Balakrishnan and Thomas Reps. Recency-abstraction for heap-allocated storage. In *International Static Analysis Symposium*, pages 221–239. Springer, 2006.

[17] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, October 2014. ACM.

[18] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[19] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. Specrop: Speculative exploitation of {ROP} chains. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 1–16, 2020.

[20] Parth Bhavsar, Ilya Safro, Nidhal Bouaynaya, Robi Polikar, and Dimah Dera. Machine learning in transportation data analytics. In *Data analytics for intelligent transportation systems*, pages 283–307. Elsevier, 2017.

[21] Kaiwan N Billimoria. *Linux Kernel Programming: A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization*. Packt Publishing Ltd, 2021.

[22] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[23] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[24] Steve Bousquet. Criminal charges filed in hacking of florida elections websites. `http://www.miamiherald.com/news/politics-government/article75670177.html`, 2016.

[25] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. *Black Hat*, 8, 2008.

[26] Cadence. Xtensa® c and c++ compiler user's guide, 2018.

[27] Cadence. Xtensa® microprocessor programmer's guide, 2018.

[28] Cadence. Xtensa® instruction set architecture, 2019.

[29] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. {Control-Flow} bending: On the effectiveness of {Control-Flow} integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[30] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[31] Shay Chen. The web application vulnerability scanner evaluation project. `https://code.google.com/archive/p/wavsep/`, 2014.

[32] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *11th IEEE Symposium on Computers and Communications*, ISCC '06, Washington, DC, USA, 2006. IEEE.

[33] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[34] Ang Cui, Michael Costello, and Salvatore J Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *20th Annual Network & Distributed System Security Symposium*, pages 1–13, 2013.

[35] Ryan Dahl. Node.js.

[36] Al Daniel. cloc: Count lines of code. `https://github.com/AlDanial/cloc`, 2019.

[37] Sanjeev Das, Bihuan Chen, Mahintham Chandramohan, Yang Liu, and Wei Zhang. Ropsentry: Runtime defense against rop attacks using hardware performance counters. *Computers & Security*, 73:374–388, 2018.

[38] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2019.

[39] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Return-oriented programming without returns on arm. Technical report, Technical Report HGI-TR-2010-002, Ruhr-University Bochum, 2010.

[40] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51, 2011.

[41] Ruan De Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (cfi). *arXiv preprint arXiv:1706.07257*, 2017.

[42] Christian DeLozier, Kavya Lakshminarayanan, Gilles Pokam, and Joseph Devietti. Hurdle: Securing jump instructions against code reuse attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 653–666, 2020.

[43] Mohamed Elsabagh, Daniel Barbara, Dan Fleck, and Angelos Stavrou. Detecting rop with statistical learning of program characteristics. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 219–226, 2017.

[44] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict virtual call integrity checking for c++ binaries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 140–154. ACM, 2017.

[45] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*, Berkeley, CA, USA, 2010. USENIX Association.

[46] K Virgil English, Islam Obaidat, and Meera Sridhar. Exploiting memory corruption vulnerabilities in connman for iot devices. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 247–255. IEEE, 2019.

[47] Exploit Database. Offensive Security's Exploit Database Archive. `https://www.exploit-db.com`, 2019.

[48] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013.

[49] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 474–484, USA, 2009. IEEE Computer Society.

[50] Matheus Eduardo Garbelini. Easily crashing esp8266 wi-fi devices, 2019.

[51] Jeff Goldman. Researchers find russian hacker selling access to u.s. election assistance commission. `https://www.esecurityplanet.com/hackers/researchers-find-russian-hacker-selling-access-to-u.s.-election-assistance-commission.html`, 2016.

[52] Google. Error-prone: Catch common java mistakes as compile-time errors. `https://github.com/google/error-prone`, 2019.

[53] Khushi Gupta and Tushar Sharma. Changing trends in computer architecture: A comprehensive analysis of arm and x86 processors. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, pages 619–631, 06 2021.

[54] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

[55] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14*, pages 175–185, New York, NY, USA, 2006. ACM.

[56] Sarah Heckman, Kathryn T. Stolee, and Christopher Parnin. 10+ years of teaching software engineering with itrust: The good, the bad, and the ugly. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, pages 1–4, New York, NY, USA, 2018. ACM.

[57] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. *WOOT*, 12:64–76, 2012.

[58] Matthias Höschele and Andreas Zeller. Mining input grammars with AUTOGRAM. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 31–34, 2017.

[59] Jin-bing Hou, Tong Li, and Cheng Chang. Research for vulnerability detection of embedded system firmware. *Procedia Computer Science*, 107:814–818, 2017.

[60] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. Revealing injection vulnerabilities by leveraging existing tests. In *Proceedings of the International Conference on Software Engineering*, 2020.

[61] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. Revealing injection vulnerabilities by leveraging existing tests (artifact). 2020.

[62] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. Revealing injection vulnerabilities by leveraging existing tests (github). `https://github.com/gmu-swe/rivulet`, 2020.

[63] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[64] Xin Huang, Fei Yan, Liqiang Zhang, and Kai Wang. Honeygadget: A deception based approach for detecting code reuse attacks. *Information Systems Frontiers*, pages 1–15, 2020.

[65] Zhi Jun Huang, Tao Zheng, and Jia Liu. A dynamic detective method against rop attack on arm platform. In *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*, pages 51–57. IEEE, 2012.

[66] Zi-Shun Huang and Ian G Harris. Return-oriented vulnerabilities in arm executables. In *2012 IEEE Conference on Technologies for Homeland Security (HST)*, pages 1–6. IEEE, 2012.

[67] iTrust Team. itrust - github. `https://github.com/ncsu-csc326/iTrust`, 2019.

[68] Sadeeq Jan, Cu D. Nguyen, and Lionel C. Briand. Automated and effective testing of web services for xml injection attacks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 12–23, New York, NY, USA, 2016. ACM.

[69] Jenkins Project Developers. Jenkins. `https://jenkins.io`, 2019.

[70] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[71] Jonathan Hedley. jsoup: Java html parser. `https://jsoup.org/`, 2019.

[72] JSqlParser Project Authors. Jsqlparser. `http://jsqlparser.sourceforge.net/`, 2019.

[73] Prakasam Kannan, Thomas Austin, Mark Stamp, Tim Disney, and Cormac Flanagan. Virtual values for taint and information flow analysis. 2016.

[74] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: A static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[75] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. *ACM SIGARCH Computer Architecture News*, 40(3):94–105, 2012.

[76] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[77] Adam Kieżun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE 2009, Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, Vancouver, BC, Canada, May 2009.

[78] Tracy Kitten. Card fraud scheme: The breached victims. `http://www.bankinfosecurity.com/card-fraud-scheme-breached-victims-a-5941`, 2013.

[79] George T. Klees, Andrew Ruef, Benjamin Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

[80] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale javascript applications (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 541–551. IEEE, 2015.

[81] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[82] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, volume 10, 2012.

[83] Yongje Lee, Ingoo Heo, Dongil Hwang, Kyungmin Kim, and Yunheung Paek. Towards a practical solution to detect code reuse attacks on arm mobile devices. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2015.

[84] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. Integration of rop/jop monitoring ips in an arm-based soc. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 331–336. IEEE, 2016.

[85] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. Using coresight ptm to integrate cra monitoring ips in an arm-based soc. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(3):1–25, 2017.

[86] Kai Lehniger, Marcin Aftowicz, Peter Langendörfer, and Zoya Dyka. Challenges of return-oriented-programming on the xtensa hardware architecture. In *EUROMICRO Conference on Digital System Design*, pages 1–6, 2020.

[87] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1193–1204, New York, NY, USA, 2013. ACM.

[88] Ben Livshits. Defining a set of common benchmarks for web application security. In *Proceedings of the Workshop on Defining the State of the Art in Software Security Tools*, 01 2005.

[89] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 385–398, New York, NY, USA, 2013. ACM.

[90] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[91] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 71–76, 2011.

[92] Ganapathy Mani, Vikram Pasumarti, Bharat Bhargava, Faisal Tariq Vora, James MacDonald, Justin King, and Jason Kobes. Decrypto pro: Deep learning based cryptomining malware detection using performance counters. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 109–118. IEEE, 2020.

[93] Wes Masri and Andy Podgurski. Using dynamic information flow analysis to detect attacks against applications. In *Proceedings of the 2005 Workshop on Software Engineering for Secure Systems&Mdash;Building Trustworthy Applications*, SESS '05, pages 1–7, New York, NY, USA, 2005. ACM.

[94] Nicolò Maunero, Paolo Prinetto, Gianluca Roascio, and Antonio Varriale. A fpga-based control-flow integrity solution for securing bare-metal embedded systems. In *2020 15th Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–10. IEEE, 2020.

[95] Rick Miller. "foreign" hack attack on state voter registration site. `http://capitolfax.com/2016/07/21/foreign-hack-attack-on-state-voter-registration-site/`, 2016.

[96] M. Mohammadi, B. Chu, and H. R. Lipford. Detecting cross-site scripting vulnerabilities through automated unit testing. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 364–373, July 2017.

[97] Lou Montulli and David M. Kristol. HTTP State Management Mechanism. RFC 2965, October 2000.

[98] National Institute of Standards and Technology. Juliet test suite for java. `https://samate.nist.gov/SRD/testsuite.php`, 2017.

[99] National Vulnerability Database. Cve-2017-5638 detail. `https://nvd.nist.gov/vuln/detail/CVE-2017-5638`, 2017.

[100] National Vulnerability Database. National vulnerability database search for "execute arbitrary commands". https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&query=execute+arbitrary+commands&search_type=all, 2022.

[101] University of Maryland. Findbugs - find bugs in java programs. http://findbugs.sourceforge.net, 2019.

[102] Hyunyoung Oh, Yeongpil Cho, and Yunheung Paek. A metadata-driven approach to efficiently detect code-reuse attacks on arm multiprocessors. *The Journal of Supercomputing*, 77(7):7287–7314, 2021.

[103] Hyunyoung Oh, Myonghoon Yang, Yeongpil Cho, and Yunheung Paek. Actimon: Unified jop and rop detection with active function lists on an soc fpga. *IEEE Access*, 7:186517–186528, 2019.

[104] Adebayo Omotosho, Gebrehiwet B Welearegai, and Christian Hammer. Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 510–519, 2022.

[105] Open Web Application Security Project. Owasp top 10 - 2017 the ten most critical web application security risks. https://www.owasp.org/index.php/Top_10-2017_Top_10, 2017.

[106] Open Web Application Security Project. Expression language injection. https://www.owasp.org/index.php/Expression_Language_Injection, 2019.

[107] Open Web Application Security Project. Owasp benchmark project. https://www.owasp.org/index.php/Benchmark, 2019.

[108] Open Web Application Security Project. Testing for sql wildcard attacks (owasp-ds-001). https://www.owasp.org/index.php/Testing_for_SQL_Wildcard_Attacks_(OWASP-DS-001), 2019.

[109] Open Web Application Security Project. Xss filter evasion cheat sheet. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet, 2019.

[110] OW2 Consortium. Asm. https://asm.ow2.io/, 2019.

[111] Hilmi Ozdoganoglu, TN Vijaykumar, Carla E Brodley, Benjamin A Kuperman, and Ankit Jalote. Smashguard: A hardware solution to prevent security attacks on the function return address. *IEEE Transactions on Computers*, 55(10):1271–1285, 2006.

[112] Vivek Parikh and Prabhaker Mateti. Aslr and rop attack mitigations for arm-based android devices. In *International Symposium on Security in Computing and Communication*, pages 350–363. Springer, 2017.

[113] Changhee Park, Hongki Lee, and Sukyoung Ryu. All about the with statement in javascript: Removing with statements in javascript applications. *ACM SIGPLAN Notices*, 49(2):73–84, 2013.

[114] Changhee Park and Sukyoung Ryu. Scalable and precise static analysis of javascript applications via loop-sensitivity. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[115] Jihyeok Park, Xavier Rival, and Sukyoung Ryu. Revisiting recency abstraction for javascript: towards an intuitive, compositional, and efficient heap abstraction. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 1–6, 2017.

[116] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. Analysis of javascript web applications using safe 2.0. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 59–62. IEEE Press, 2017.

[117] David Pfaff, Sebastian Hack, and Christian Hammer. Learning how to prevent return-oriented programming efficiently. In *International Symposium on Engineering Secure Software and Systems*, pages 68–85. Springer, 2015.

[118] Aravind Prakash and Heng Yin. Defeating rop through denial of stack pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 111–120, 2015.

[119] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[120] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, February 2017.

[121] IBM Research. Wala the tj watson libraries for analysis.

[122] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2010.

[123] Sukyoung Ryu, Alexander Jordan, and Dongsun Kim. Safe tutorial: Taint analysis for web applications.

[124] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[125] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. Security and privacy challenges in industrial internet of things. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[126] Tejas Saoji, Thomas H Austin, and Cormac Flanagan. Using precise taint tracking for auto-sanitization. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 15–24, 2017.

[127] Tejas Saoji, Thomas H. Austin, and Cormac Flanagan. Using precise taint tracking for auto-sanitization. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, pages 15–24, New York, NY, USA, 2017. ACM.

[128] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. *Acm Sigplan Notices*, 48(6):165–174, 2013.

[129] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[130] Matthew Schwartz. Equifax's data breach costs hit $1.4 billion. `https://www.bankinfosecurity.com/equifaxs-data-breach-costs-hit-14-billion-a-12473`, 2019.

[131] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[132] Ashwin Seshagiri. How hackers made $1 million by stealing one news release. `https://www.nytimes.com/2015/08/12/business/dealbook/how-hackers-made-1-million-by-stealing-one-news-release.html?_r=0`, 2015.

[133] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[134] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[135] Dimitris E. Simos, Jovan Zivanovic, and Manuel Leithner. Automated combinatorial testing for detecting sql vulnerabilities in web applications. In *Proceedings of the 14th International Workshop on Automation of Software Test*, AST '19, pages 55–61, Piscataway, NJ, USA, 2019. IEEE Press.

[136] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[137] Petr Somol, Jana Novovicová, Jirí Grim, and Pavel Pudil. Dynamic oscillating search algorithm for feature selection. In *2008 19th International Conference on Pattern Recognition*, pages 1–4. IEEE, 2008.

[138] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *CCS '13*, New York, NY, USA, 2013. ACM.

[139] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. Static identification of injection attacks in java. *ACM Trans. Program. Lang. Syst.*, 41(3):18:1–18:58, July 2019.

[140] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: Taint Analysis of Framework-based Web Applications. In *OOPSLA '11*. ACM, 2011.

[141] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: Taint Analysis of Framework-based Web Applications. In *OOPSLA '11*. ACM, 2011.

[142] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. *ECOOP 2012–Object-Oriented Programming*, pages 435–458, 2012.

[143] Derek Staahl. Hack that targeted arizona voter database was easy to prevent, expert says. `http://www.azfamily.com/story/32945105/hack-that-targeted-arizona-voter-database-was-easy-to-prevent-expert-say` 2016.

[144] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Understanding and automatically preventing injection attacks on node. js. Technical report, Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science, 2016.

[145] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[146] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

[147] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, pages 85–96, New York, NY, USA, 2004. ACM.

[148] Kwangwon Sun and Sukyoung Ryu. Analysis of javascript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)*, 50(4):1–34, 2017.

[149] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 77–91, Hollywood, CA, 2012. USENIX.

[150] Terence Parr. Antlr. `https://www.antlr.org/`, 2019.

[151] The Apache Software Foundation. Ognl - apache commons ognl - developer guide. `https://commons.apache.org/proper/commons-ognl/developer-guide.html`, 2019.

[152] The Apache Software Foundation. Security. `https://struts.apache.org/security/`, 2019.

[153] The Eclipse Foundation. Jetty - servlet engine and http server. `https://www.eclipse.org/jetty/`, 2019.

[154] The MITRE Corporation. Cwe-601: Url redirection to untrusted site ('open redirect'). `https://cwe.mitre.org/data/definitions/601.html`, 2019.

[155] Julian Thomé, Alessandra Gorla, and Andreas Zeller. Search-based security testing of web applications. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, SBST 2014, pages 5–14, New York, NY, USA, 2014. ACM.

[156] Thotcon. The complete esp8266 psionics handbook, 2016.

[157] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI '09*, pages 87–97, New York, NY, USA, 2009. ACM.

[158] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI '09*, pages 87–97, New York, NY, USA, 2009. ACM.

[159] Gordon Mah Ung. Everything you wanted to know about amd's new trueaudio technology, 2013.

[160] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689, 2017.

[161] Yap Bee Wah, Hezlin Aryani Abd Rahman, Haibo He, and Awang Bulgiba. Handling imbalanced dataset using svm and k-nn approach. In *AIP Conference Proceedings*, volume 1750, page 020023. AIP Publishing LLC, 2016.

[162] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Trans. Inf. Syst. Secur.*, 14(2), September 2011.

[163] Xueyang Wang and Jerry Backer. Sigdrop: Signature-based rop detection using hardware performance counters. *arXiv preprint arXiv:1609.02667*, 2016.

[164] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, and Guimin Zhang. A survey of exploitation techniques and defenses for program data attacks. *Journal of Network and Computer Applications*, 154:102534, 2020.

[165] Vincent M Weaver and Sally A McKee. Can hardware performance counters be trusted? In *2008 IEEE International Symposium on Workload Characterization*, pages 141–150. IEEE, 2008.

[166] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346, 2013.

[167] Shiyi Wei and Barbara G Ryder. Adaptive context-sensitive analysis for javascript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[168] Gebrehiwet B Welearegai, Max Schlueter, and Christian Hammer. Static security evaluation of an industrial web application. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1952–1961, 2019.

[169] Gebrehiwet Biyane Welearegai and Christian Hammer. Idea: Optimized automatic sanitizer placement. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 87–96, Cham, 2017. Springer International Publishing.

[170] Gebrehiwet Biyane Welearegai, Chenpo Hu, and Christian Hammer. Detecting and preventing rop attacks using machine learning on arm. In *Submitted to 47th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE*, 2013.

[171] Chris Williams. Microsoft's hololens secret sauce: A 28nm customized 24-core dsp engine built by tsmc, 2016.

[172] World Wide Web Consortium. Html 5.2. `https://www.w3.org/TR/html52/`, 2017.

[173] World Wide Web Consortium. Parsing html documents. `https://html.spec.whatwg.org/multipage/parsing.html`, 2019.

[174] Infosec Writers. Bypassing non-executable-stack during exploitation using return-to-libc.

[175] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 595–605, Piscataway, NJ, USA, 2012. IEEE Press.

[176] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Trans. Softw. Eng. Methodol.*, 23(4):32:1–32:28, September 2014.

[177] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 457–468, 2018.

[178] HongWei Zhou, Xin Wu, WenChang Shi, JinHui Yuan, and Bin Liang. Hdrop: Detecting rop attacks using performance monitoring counters. In *International Conference on Information Security Practice and Experience*, pages 172–186. Springer, 2014.