NANYANG TECHNOLOGICAL UNIVERSITY

Hardware-Assisted Online Defense Against Malware and Exploits

Sanjeev Kumar Das

School of Computer Science and Engineering

A thesis submitted to Nanyang Technological University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

November, 2016

THESIS ABSTACT

Hardware-Assisted Online Defense Against Malware and Exploits

by

Sanjeev Kumar Das

Doctor of Philosophy

School of Computer Science and Engineering

Nanyang Technological University, Singapore

Security is a major concern in the computing systems with the increasing number of cyber attacks in recent years. Mainstream security solutions (e.g., antivirus, scanners) are mostly implemented as software. Thus, the vulnerabilities in these solutions can be exploited to disable or bypass the defense, like rootkit and bootkit. Besides, software approaches suffer from the high performance overhead and resource requirement. As the result, they can only be implemented partially, which leaves opportunities for the adversaries to exploit the system.

Recently, hardware-assisted solutions for cyber security have emerged as a promising protection against the evolving attacks. Compared to the software solutions, hardware-based solutions have several advantages. First, they are difficult to be bypassed or identified by malware as they are running below the operating system. Second, hardware supported security approaches are much more energy- and power-efficient, which are desirable for runtime defense. Third, hardware-based implementation offers unmatched visibility into the program execution, which provides opportunities to develop novel security techniques. Finally, the implementation in hardware inherently offers the quick detection of attacks. It runs separately from the processor and hence, barely affects the processor performance.

In this thesis, we study hardware-assisted security approaches to defend against cyber attacks at runtime. We focus on two critical issues of software security: malware and exploits. Malware (short for "**mal**icious soft**ware**") is a collective term for any program that enters into a system without the knowledge of the user and deliberately fulfills the harmful intent of an attacker. An exploit refers to a software program that attacks the system by taking advantage of a vulnerability present in the system. Adversary commonly uses exploits to attack an operating system or application vulnerability to gain privileges, so that they can run malicious code in the system. Hence, the defense at the exploitation stage

can eliminate the threats at the point of pre-infection and protect from malware download and its execution.

We begin with analyzing malware and proposing detection solutions.

1. First, we propose a hardware-enhanced architecture to detect malware at runtime. Our approach aims to capture the malicious features (i.e., high-level semantics) of malware. We develop a machine learning approach in FPGA to train a classifier using features obtained from the known samples. At runtime, the trained classifier is used to classify the unknown samples as malware or benign, with early prediction. The proposed machine learning approach can effectively detect unknown malware samples based on the trained features and this approach is highly scalable. There are two challenges for this approach: 1) it can be defeated if features in the machine learning method are not comprehensive or if malware uses a new attacking strategy; and 2) the machine learning approach cannot explain the attack behavior of malware and hence fails for malware classification.

2. Second, we propose a semantic-based malware detection method to address the two challenges above. We firstly propose to use deterministic finite automata (DFA) to model the malware behavior. Then we learn the attack model of malware via an offline analysis. At runtime, we implement a DFA-based detection approach in hardware to check whether a program's execution contains the malicious behavior specified in the DFA. This approach can effectively capture the attack behavior of malware and has the potential to detect zero-day malware. Implemented in hardware, our architecture offers a real-time detection with low performance and resource overhead. More importantly, it cannot be bypassed by malware using sophisticated evasive techniques.

To have a fine-grained understanding of malware, we investigate the common exploitation techniques in the second half of the thesis.

1. First, we present a fine-grained control flow integrity approach to defend against runtime memory attacks using a hardware-enhanced architecture. Based on the offline profiling of the original benign program, our security model can effectively detect memory attacks (injected in the original program) at runtime, which works in parallel to the processor with low performance (<1%) and area overhead (0.02%). This approach requires profiling of

the normal program behavior, which may not be feasible for all applications, and also a minor modification in the existing CPU architecture.

2. Second, we propose a more practical and lightweight runtime defense approach against common exploitation techniques. This approach leverages low-level hardware features of the commodity processors (i.e., hardware performance counter) to detect the exploit behavior at runtime. The advantage of this approach is twofold: first, it can be implemented in the existing systems without any hardware modification; second, in addition to the low performance overhead, it can precisely detect exploits with high accuracy and low false positive rate.

Acknowledgements

First and foremost, I would like to express my deep gratitude to my advisors Prof. Yang Liu and Prof. Wei Zhang, for their guidance, advice, and continuous support throughout my Ph.D. study. They have always been appreciative of new ideas and encouraged critical thinking and technical writing, which have helped me improve my skills. They have been of invaluable support throughout these years in many ways, which has really helped me to shape my research.

I am grateful to Prof. Chang Chip Hong and Prof. Alwen Tiu, who were in my Thesis Advisory Committee, for their valuable suggestions and comments on my research works.

I am grateful to Mahintham Chandramohan, Dr. Bihuan Chen, Xiao Hao, Dr. Wei Lei, and Dr. Yinxing Xue for their valuable advice, suggestions and brainstorming sessions that helped me to improve my research work.

I would like to express appreciation for my friends and colleagues at the Cyber Security Lab (CSL) for their support and encouragement, especially, Guozhu Meng, Junjie Wang, Xiaoning Du, HongXu Chen, Ruitao Feng and Fei Xiao. I take this opportunity to thank our laboratory executive Tan Suan Hai for providing technical support at CSL.

I would like to thank my friends and colleagues at Hardware & Embedded Systems Lab (HESL), especially Suman Deb, Abhishek Jain, Deheng, Vijeta, Supriya, Ronak, Hui Yan, Rakesh, Yingnan, Abhishek Ambede, and Sumedh for their support. I am also thankful to Chua Ngee Tat for his technical assistance in HESL.

I would like to take this opportunity to thank my friend Yang Liwei, who has always supported me with his advice and suggestions when needed. I would also like to thank my close friends – Anjali, Suman, Assel and Ajin, with whom I had a wonderful time in Singapore.

Finally, I am indebted to my parents for always encouraging me for my studies and raising me to the level where I stand today. I am extremely thankful to my sweetheart Indira, for all the love, motivation and encouragement in these years.

Contents

List of Figures

List of Tables

1

Introduction

Security is a serious concern in computing systems because of the ever-increasing number of attacks in recent years. These systems contain the amplitude of private and critical information, as we rely heavily on them for many important aspects of our life, including communications, transportation, finance, medication and so on. With the increasing complexity, connectivity and ubiquity of computing systems, the vulnerabilities and attacks are also scaling up. Consequently, the security breaches result in an unprecedented amount of damages.

Recently, the number and the level of sophistication of cyber security threats – including vulnerabilities, exploits and malware – have grown up, which suggest that we must remain vigilant about securing the computing systems. Symantec reports that more than 430 million new malware samples were discovered in 2015, which account for 36% increase compared to the year before [1]. Malware has

also skyrocketed in the mobile devices, such as smartphones and tablets. A recent McAfee report [2] shows that the number of unique malware targeting mobile operating systems, including tablets that are running Android and iOS, tripled in 2 years, i.e., from 4 million in the beginning of 2014 to >12 million by the end of 2015. Adversary uses new attack methods continuously to defeat existing defense mechanisms. The security threats will continue to proliferate as the attack techniques keep evolving on the regular basis. Besides, it requires little effort to automate the hacker tools, which allows more adversaries to perform such attacks. Many recent attacks show that a significant amount of damage can be done, if proper countermeasures are not taken in time.

The newspaper headlines are often buzzed with attacks and data breaches:

- *Attack on Anthem:* In 2015, zero-day exploits and custom-developed malware were used to perform a major data breach, which exposed 78 million patient records at Anthem, a health care company in the US [1].
- *Attack on banks:* Adversaries used Carbanak malware to infiltrate into the bank administrator's computers, to steal billions of dollars from the banks [3].
- *Lethal attacks on medical devices:* Remote attacks on insulin pumps, pacemakers and implanted defibrillators were performed by injecting fake data through wireless signals [4].
- *Attack on industrial control systems:* Stuxnet worm subverted a microcontroller system connected to a proprietary network in a locked-down facility in Iran; using conventional malware techniques such as phishing emails and USB sticks to reach its intended target [5].
- *Misfortune cookie vulnerability:* Researchers found a vulnerability affecting at least 12 million leading-brand home and small office routers [6]; exploitation of which could allow to monitor all the sensitive and private data traveling through the gateway and to spread malware.

- *ATM Jackpotting:* ATM exploits allowed to spit out money on demand and record sensitive data from ATM cards. The attacks were performed using a malware-loaded USB and also remotely over a network [7].

- *Remote attacks on vehicle*: Recently, security researchers exploited a vulnerability in car's infotainment system to hack into other systems. They were able to track a vehicle's exact location, turn the lights/blinkers on or off, tamper brakes and steering [8].

Over the years, the motivation of the hackers has changed from technical-skill show-off to the monetary gains. Reports suggest that there are black markets which offer opportunities to trade private and sensitive information – such as credit cards, login details of a sensitive machine [9]. There are also state-funded agencies which develop malware to target specific systems for espionage or sabotage [10, 11]. The arms race between the adversaries and the security personnel has already led to the development of extremely sophisticated threats. On the other hand, because of the driving forces like Internet of Things (IoT), many computing systems are connected to each other. Given the wider connectivity, the adversaries get a larger surface to perform the attack, which can result in an unprecedented amount of damages. Thus, greater security countermeasures are needed to prevent the modern computing systems from being attacked.

1.1 Malware and Exploits

Malware ("malicious software") is the major threat to the computing systems. In recent years, the volume of malware has increased significantly. In the arms race with anti-virus solutions, malware has evolved from simple ones to the complex ones with different forms – virus, worm, trojan, adware, spyware, backdoor, botnet, and rootkit [12]. They use several channels to penetrate into the system, e.g., exploiting vulnerabilities, abusing application stores, clicking on links, phishing emails, repackaging into normal applications, and through USB, memory card and browser utilities. Zero-day and unknown malware is critical threats as

they are undetected by traditional anti-virus engines. Sandboxing techniques are employed to detect these threats by looking for malicious activities at OS-level once the malware is active. In response, the advanced hackers design new evasive techniques to stay undetected. Though these detection techniques are effective but they are not enough anymore to prevent the system from malicious attacks.

Adversaries exploit software vulnerabilities to penetrate into the system, and then download and execute the malware. There are thousands of vulnerabilities and millions of malware programs, but there are limited methods that attackers utilize to exploit the vulnerabilities. Hackers exploit software vulnerabilities at the CPU-level. Several exploitation techniques exist today – stack pivoting, privilege escalation, memory disclosure, operating system (OS) mitigation bypass. The most popular technique is return oriented programming (ROP), which is a type of code reuse attacks. ROP hijacks small pieces of legitimate code from the memory and manipulates the CPU to download and execute the actual malware. Detecting this manipulation at the CPU-level is the key to stop malicious attacks before they even happen. Thus, the defense at the exploitation stage can eliminate the threats at the point of pre-infection and protect from malware downloads.

1.2 Motivation

Mainstream attack detection solutions, e.g., antivirus and scanners, are mostly implemented as software, which have the same level of vulnerability as any other software. Thus, the vulnerabilities in these solutions can be exploited to disable or bypass the defense. On the other hand, hackers develop advanced evasive strategies to hide malicious behavior and remain dormant in the presence of anti-virus tools. For example, the malware uses anti-debugger techniques – anti-ptrace, anti-sigtrap and anti-breakpoint – to detect the presence of malware detectors in the host and exits cleanly once it finds them [13]. Most importantly, software approaches are infeasible for sophisticated classes of malware; such as rootkits and bootkits, which achieve their malicious intents by infecting the kernel [14]. Since the rootkits have

the similar privilege as the OS (Ring 0), they can disable and evade the software protection. Besides, software approaches suffer from the substantial performance overhead and resource requirement. As the result, they can only be implemented partially, which leaves opportunities for the adversaries to exploit the system.

We list several other challenges that are faced by existing malware and exploit detection methods as follows:

1. **Obfuscation techniques:** Malware uses many obfuscation techniques, such as encryption, dead-code insertion, subroutine reordering, instruction substitution, code transposition, which make it harder to understand, in order to evade the conventional solutions such as antivirus and scanners [15].

2. **Anti-debugging techniques:** Recently, malware writers use advanced techniques such as logic bombs, anti-virtual machine inspection and anti-debugging techniques [12, 16] to hide the malicious activity of the malware to evade the detection.

3. **Evolving exploitation techniques:** Adversaries use advanced exploitation techniques (e.g., code reuse attacks) to bypass or disable the existing mitigations and defense techniques. Current anti-malware tools lack the ability to defend these ever-evolving exploitation techniques.

4. **High resources requirement:** Some malware detection approaches demand a large number of resources. For example, system call based techniques using n-gram, m-bag and k-tuple suffer from the huge feature space and thus require a lot of computational resources. Similarly, dependence and graph based approaches are highly compute-intensive.

5. **High performance overhead:** Many malware detection approaches are compute-intensive, thus, they can be only used for offline analysis. However, they will cause significant overhead if they are implemented at runtime.

Recently, hardware-assisted architectures for software security have emerged as a promising solution for protecting from the evolving attacks. Hackers exploit

software vulnerabilities at the CPU-level, by bypassing the existing OS-level mitigation. Therefore, designing hardware-based solutions can be more effective to defend against these attacks, as they are immutable and non-bypassable. At the same time, with the exponential increase of transistors on-chip, there is an increasing willingness among the CPU manufacturers to build security mechanisms in hardware [17–19]. Compared to the existing software solutions, hardware-based solutions have several advantages. First, they are difficult to be bypassed or identified by malware as they are running below OS. Second, hardware supported security approaches can be much more energy- and power- efficient, which are desirable for runtime defense. Third, hardware-based implementation offers unmatched visibility into the program execution, which provides opportunities to develop novel security techniques. Finally, the implementation in hardware inherently offers the quick detection of attacks. It runs separately from the processor and hence, barely affects the processor performance. Meanwhile, hardware attacks are not common as they require much more efforts and cost than the software attacks. This is because, first, it requires the physical access to the hardware components and second, the data flow is complex at the hardware level. On the other hand, hardware-assisted security solutions may require modification in the existing hardware, however, with a small modification in the hardware, billions of lines of software can be protected over its lifetime.

1.3 Research Goals

Hardware-based solution is a promising direction to build an online defense against malware and exploits. However, not all the techniques can be implemented in the hardware, because hardware is inherently less flexible as compared to software, and also has limited resources. Therefore, hardware-based approaches once built must be resistant to the attacks for a longer time. On top, it should be effective to defend against zero-day and unknown attacks. In this thesis, we make an attempt to answer the following research questions:

1. How do we build an online malware detection engine in hardware?
2. How can we detect malware variants and zero-day malware at runtime?
3. How can we build a security module in hardware to defend against runtime memory exploits?
4. How do we build a practical exploit defense module that can be adopted in the real-world settings?

To address these questions, we propose four different approaches. All the four approaches use hardware features to build online defenses against malware and exploits. To answer the first question, we build a malware detection engine using a hardware-enhanced architecture based on the machine learning technique. Second, in order to precisely capture different malware variants and zero-day malware, we propose an attack behavior modeling approach. We implement both these approaches in FPGA, which provides the flexibility to update the malware detection engine, in order to incorporate the new malicious behavior. To answer the third question, we implement a control flow integrity (CFI) checking mechanism in hardware to detect runtime memory exploits, using information obtained from an offline analysis. We show that our approach can detect runtime exploits with a very low performance overhead. Finally, we build a more practical and lightweight defense against common exploits using the available CPU-level hardware performance counters.

1.4 Summary of Contributions

This thesis shows how to design security defense mechanisms to detect malware and exploits at runtime using hardware-enhanced architecture. The contributions of this thesis are summarized as follows:

1. **GuardOL:** first, we demonstrate a hardware-enhanced architecture, called *GuardOL* (short for **Guard O**n**L**ine), to detect malware at runtime. Our

approach aims to capture the malicious features (i.e., high-level semantics) of malware. We develop a machine learning approach in FPGA to train a classifier using these features obtained from the known samples. At runtime, the trained classifier is used to classify the unknown samples as malware or benign, with early prediction. Our experimental results show that GuardOL can achieve high classification accuracy, fast detection, low power consumption and flexibility for easy functionality upgrade to adapt to new malware samples. One of the main advantages of our design is the support of early prediction – it can detect 47% of malware samples within first 30% of their execution, while 98% of the samples after their execution, with $< 3\%$ false positives. GuardOL has negligible area overhead (0.003%) and no performance overhead. The proposed machine learning approach can effectively detect unknown malware samples based on the trained features and this approach is highly scalable; however, it cannot model the attack behavior of malware. Thus, it can be defeated by adversaries by modifying the known malicious behavior.

2. **Malguard:** second, we propose a Deterministic Finite Automaton (DFA) based approach to learn the attack model of malware during offline. At runtime, we implement a DFA-based detection approach in hardware, called *Malguard* (meaning **Mal**ware **Guard**), to check whether a program execution contains the malicious behavior specified in the DFA. This approach can effectively capture the attack behavior of malware and has the potential to detect zero-day malware. Our evaluation shows that *Malguard* can recognize malware variants of the same family. Implemented in hardware, our architecture offers a real-time detection with low performance and resource overhead. More importantly, it cannot be bypassed by malware using sophisticated evasive techniques.

3. **BB-CFI:** third, we present an approach to enforce the Control Flow Integrity (CFI) at a basic block level, called *BB-CFI* (meaning **B**asic **b**lock-**CFI**), which aims to defend against runtime memory attacks using hardware-enhanced architecture. Based on the offline profiling of the programs, our

security module can effectively detect memory attacks at runtime, with zero false positives. The security module works in parallel to the processor and does not stall the processor, thus causing no significant performance penalty on the system. It has low performance (<1%) and area overhead (0.02%) on the processor. This approach requires profiling of the normal program behavior, which may not be feasible for all applications, and also a minor modification in the existing CPU architecture.

4. **ROPSentry:** fourth, we propose a more practical and novel defense framework, called *ROPSentry*, to detect return oriented programming (ROP) attacks at runtime. Based on the observation that ROP programs trigger different hardware events than normal programs generated by compilers, we leverage hardware performance counters (HPCs) to track these hardware events in order to capture the patterns of heap spray (i.e., a common ROP payload delivery mechanism) and ROP exploits. The advantage of this approach is twofold: first, it can be implemented in the existing systems without any hardware modification; second, in addition to the low performance overhead (<1%), it can effectively detect ROP exploits with high accuracy and low false positive rate.

The comparison of these four approaches is shown in Table 1.1. The first two works aim to detect malware at runtime, while last two works focus on exploit detection. *GuardOL* is a scalable approach and can detect known malware samples, however, it lacks the capability to detect unknown malware variants. *Malguard* overcomes this limitation by modeling attack behavior of malware. In addition to the detection, it is also capable of classifying malware samples based on their attack behavior. While *BB-CFI* can precisely enforce CFI to detect runtime attacks, however, it requires the offline profiling of the program. On the other hand, *ROPSentry* does not require the offline profiling and can protect against runtime exploits. A more comprehensive comparison and discussion can be found in Table 7.1 in Chapter 7.

Table 1.1: Comparison of our approaches

	GuardOL	Malguard	BB-CFI	ROPSentry
Target attacks	General malware - virus, trojan, worm, flooder, rootkit	General malware, variants and zero-day malware	Control flow attacks – stack smashing and code reuse attacks	Heap spray and ROP attacks
Approach	Feature-based machine learning approach	Attack-behavior model in the form of DFA	Control flow integrity at basic block level	Performance counters based approach
Methodology	Features extraction to train machine learning classifier for runtime classification	Malware attack behavior modeling in the form of DFA using L* algorithm	Runtime verification of branch targets based on offline profiling of basic blocks	Return miss based and Self-adaptive sampling of hardware performance counters
Limitations	Cannot detect malware variants and zero-day malware	Lacks scalability – malware attack behavior should be known	Offline profiling must cover all the valid control flow paths	Analyzer can be disabled if OS is compromised

Our proposed approaches address the challenges that are listed in Section 1.2. First, all the four approaches are based on the semantics of attacks and dynamic techniques, i.e., they build their approach based on the runtime behavior of the program. Thus, they can overcome the obfuscation techniques. Second, our first three approaches are implemented in the hardware level that is below OS, thus, they cannot be recognized by malware. ROPSentry uses CPU-level features and is implemented in the OS kernel, but, it does not rely on any debugging techniques. Thus, our approaches are resistant to the anti-debugging techniques used by malware and therefore, they cannot be bypassed. Third, BB-CFI and ROPSentry aim to defend against the advanced exploitation techniques. Fourth, all of our security approaches are designed such that they require a small amount of resources and introduce a low performance overhead.

1.5 Thesis Outline

This thesis is organized as follows: In Chapter 2, we present the background and related work on the malware defense and exploit detection. Chapter 3 describes a hardware-based approach to model and detect malware at runtime, using machine learning technique. Then we present the design of an FPGA-based anti-malware engine, called *GuardOL*. In Chapter 4, we explain DFA based approach to learn

malware attack behavior using L* algorithm. Then we present hardware design of *Malguard* that aims to detect malware variants and unknown samples at runtime. In Chapter 5, we describe our BB-CFI approach and present the design of control flow checker to detect memory attacks at runtime. Then we present the limitations and experimental results of our approach. Chapter 6 presents *ROPSentry*, a runtime defense framework to protect against exploits, particularly heap spray and ROP using HPCs. We show how HPCs can be leveraged to detect these exploits at runtime, with a very low performance overhead and false positive rate. In Chapter 7, we compare our different approaches and highlight the significance of each. Finally, we conclude the thesis with our final thoughts and suggest future directions for research in this area.

1.6 Publication List

The list of publications generated from this thesis is listed as follows:

1. Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. "Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware." **IEEE Transactions on Information Forensics and Security (TIFS)**, 11, no. 2 (2016): 289-302.

2. Sanjeev Das, Wei Zhang, and Yang Liu. "A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems". **IEEE Transactions on Very Large Scale Integration Systems (TVLSI)**, 24, no. 11 (2016): 3193-3207.

3. Sanjeev Das, Chen Bihuan, Mahintham Chandramohan, Yang Liu, and Wei Zhang. "Self-Adaptive Defense against ROP Exploits using Hardware Performance Counters". **IEEE Transactions on Information Forensics and Security (TIFS)** (Under submission).

4. Sanjeev Das, Xiao Hao, Yang Liu, and Wei Zhang. "Online Malware Defense Using Attack Behavior Model". In **IEEE Int'l Symposium on Circuits & Systems (ISCAS)**, 2016.

5. Sanjeev Das, Wei Zhang, and Yang Liu. "Reconfigurable Dynamic Trusted Platform Module for Control Flow Checking." In **IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**, pp. 166-171, 2014.

6. Sanjeev Das, Wei Zhang, and Yang Liu. "FPGA Based Control Flow Checking." In **Design Automation Conference (DAC)**. 2014 (poster).

2

Background on Malware and Exploits

2.1 Malware

Malware is software that achieves deliberately the harmful intent of an attacker [20]. Though the initial motivation for malware developers was to show vulnerable points in the system, their motivation has been profit-driven due to the underground economy based on malware. Vulnerabilities and bugs in the software are unavoidable and increase as the complexity of software increases day by day. Malware exploits vulnerabilities in the trusted software, such as web browser, vulnerable services on network, spam email [12]. Malware has been steadily increasing and evolving in the recent years, as shown by the statistics [21]. Malware developers use obfuscation techniques in order to avoid being detected by traditional

anti-virus tools [15]. Hence, an effective malware defense becomes an extreme necessity in order to protect from the harmful consequences resulting from it.

2.2 Malware Classification

Malware samples are found in several forms such as *Worm, Virus, Trojan horse, Spyware, Rootkit and Bot.*

- *Worm* is a program that is prevalent in computer networks, which runs independently. It propagates itself to other machines and uses vulnerabilities of the system to perform its malicious intents [22].
- *Virus* is a malicious program that adds itself to the other programs. It depends on the host program to get activated, but cannot run independently [22]. Generally, it spreads by infecting files in the host machine, on a shared file server and other vulnerable hosts that it can infect.
- *Trojan*, a.k.a. Trojan horse, pretends to be a useful and legitimate program, but it performs malicious operations in the background. It may download other malware, change the system settings, or infect host files.
- *Spyware* is a program installed clandestinely on computers, which records sensitive information from the infected system and transfers them to the attacker.
- *Rootkit* is the program that has the ability to hide its presence from the user system. Rootkit techniques are applied by many malware, at the user-mode or kernel-level, to conceal their information about the processes, files or network connections on victim's system.
- *Bot* is the software which allows the victim's system to be controlled remotely. It is commonly used for sending spam emails or to perform spying activities.

A significant amount of work has been proposed for malware detection [12, 23–25]. Malware detection techniques can be broadly classified into two categories: static and dynamic. Static analysis refers to the technique that analyzes a program by inspection (without executing the program) while dynamic analysis refers to the technique that analyzes program behavior during the execution. We present a brief overview of static analysis and focus on dynamic analysis techniques.

2.3 Malware Detection using Static Analysis

Static analysis refers to the technique that examines malware without actually executing it. It usually employs several tools such as disassemblers, decompilers, source code analyzers to obtain a signature from the program and uses pattern matching techniques to detect malware. Signature refers to a unique pattern or fingerprint of the program executable. The patterns extracted from known malware are used for detection of malware. Since the signature is unique, the detection rate is faster and has a small error rate. Because of this, the most prevalent anti-virus tools use these techniques due to the low error rate [26]. Signature technique examines the static content of malicious binary. Static methods have an advantage of full coverage of the code and can detect malware even before the execution of the sample, which can eventually protect from malicious attempts. But, static analysis has certain limitations:

- It requires a lot of manpower and time to extract signatures of each malware.
- Obfuscated techniques employed by malware can easily evade static malware detection, such as the structure of malware can be changed without changing its behavior.
- Malware authors apply concealment strategies to hide its behavior when it gets aware of being tracked, which can defend static detection methods.
- Besides, the malware also applies defensive techniques such as encryption.

- The source code of malware is not available. Disassembly of the binary executable of the sample might result into ambiguous results, such as in the case of self-modifying code.

2.4 Malware Detection using Dynamic Analysis

Static analysis technique is scalable as it can analyze the complete program, whereas dynamic analysis technique is more accurate. Dynamic analysis is based on runtime information, which mainly focuses on the semantics of the sample program. These techniques utilize semantics such as system calls and/or their arguments, control flow graph, instruction sequences to detect malware. The detection techniques can be categorized as [12]:

- Function Call Monitoring
- Function Parameter Analysis
- Information Flow Tracking
- Instruction Trace

In the following section, we briefly summarize the techniques proposed in the different categories.

1. **Function Call Monitoring**

   Function call monitoring includes those approaches which use Application Programming Interface (API), system calls, Windows Native API (which lies between system call interface and Windows API) to model malware behavior.

   Bayer et al. proposed an automated tool TTAnalyze [27], to dynamically analyze the behavior of Windows executable by monitoring security-relevant actions in an emulated environment. The technique monitors Windows native system calls and Windows API functions invoked by the program for analysis.

Forest et al. proposed a system call based technique considering the fact that anomalies should leave relics in system calls executed by kernel [28]. System call patterns provide a rich amount of information, representing the raw interaction between the program and the host system. The author proposed system calls as the best granularity for intrusion detection systems, without considering the arguments passed to each system call. This method loses some information about the relationships between system call sequences.

Creech et al. proposed host-based anomaly detection methodology using the semantics of the discontiguous system call patterns in order to increase detection rates and reduce false alarms [29]. This approach uses the semantic structure of kernel level syscall behavior and extreme learning machine techniques in order to detect the intrusions. The author proposes the number of "discontiguous system call patterns" in each training sample as a behavior to analyze a new sample. First, the theoretical possible phrase of different length is explored. The sample is then examined for available phrase and discontiguous words count is observed. The count of discontiguous words is used for training a decision engine.

Malicious features extraction from unpacked executables for reverse obfuscation is a labour intensive work and demands a deeper understanding of low-level programming including kernel and assembly language. To solve this problem, [30] proposed an automated method of extracting API call and analyzing them in order to understand their use for the malicious purpose.

In [31], Canali et al. explored various behavior models based on: atoms (system calls with/without arguments, action with/without arguments), structures to combine atoms (*n-gram, k-tuples and m-bag*) and cardinality (number of atoms in signature, i.e. values of n, k and m). An n-gram is a sequence of n atoms that appear in consecutive order in the program execution trace, while an m-bag contains a bag of m atoms without any order and a k-tuple combines n atoms that appear in order but at any distance from each other

in program execution. Though, the detection rate is 99% for "2-bags of 2-tuples for action with arguments", the method suffers from scalability issues. The number of features generated by n-gram, m-bag or k-tuples is huge, requiring a lot of memory and time for feature extraction when the number of malware samples is large. Besides, parameter tuning also has to be done for optimal performance, which may lead to a high detection rate but also results in an overfitting problem.

AccessMiner technique takes a system-centric approach, which models the interaction between the benign programs and OS, instead of commonly used program-centric approach [32].

2. **Function Parameter Analysis**

   The function parameters are used to infer the behavior of the malware program, such as parameters being passed to the function and return values of the function can give the correlation of individual function calls.

   Chandramohan et al. proposed a scalable clustering approach to identify and group malware samples that exhibit similar behavior [33]. The author proposes a precise approach to capture a malware program's behavior. To this end, the execution of a program is monitored and its behavioral profile is created by abstracting system calls, their dependencies, and the network activities to a generalized representation consisting of OS objects and OS operations. An efficient and fast algorithm for clustering large sets of malware samples, which avoids calculating n^2 distances between all pairs of n samples, is the main contribution of this approach.

   Maggi et al. describe unsupervised host-based intrusion detection system, based on system call arguments and sequences [34]. A clustering process helps to fit models to system call arguments and creates interrelations among different arguments of a system call. Using a behavioral Markov model, the method captures time correlations and abnormal behaviors. First, anomaly detection models are built based on system call parameters. Then clustering of arguments is done to infer different ways to use same system call and also to create correlation among the different parameters of the same system call.

On the other hand, Liu et al. [35] proposed arguments passed to each system call to study the semantics of the program, rather than semantic patterns in system call traces, for behavioral analysis of malware.

[36] collected a sequence of system calls of a process and used an n-gram technique to construct malware and benign features. It uses a genetic algorithm to tune the goodness value to the common features of malware and benign programs. The goodness value estimates the likelihood of the common features being malware or benign. Unlike machine learning approaches, the genetic algorithm does not require a complete feature vector to detect malware. Hence, malware can be detected much earlier and the system can be saved from major damages. In the worst case, the whole program must be executed to detect malware using this technique.

Wang et al. proposed a hybrid approach, combining both static and dynamic analysis to achieve both scalability and accuracy, to automatically classify Javascript malware samples along with their detection [37]. Their approach uses textual analysis and function call patterns to build the attack model of the Javascript malware so that it can potentially detect new malware variants and new vulnerabilities. To automatically learn the attack behaviors of malware, Xue et al. proposed to use Deterministic Finite Automaton (DFA) [38]. They used a data dependency analysis, defense rules and Javascript replay mechanism to identify the malicious trace.

3. **Information Flow Tracking**

   The control and data flow based techniques are summarized in this category.

   [39] focuses on malware detection on the android system, using function call graphs in order to combat the obfuscation techniques at the instruction level. Identification of similarities in graphs is a non-trivial task. This approach uses machine learning classification of graphs, by efficient embedding of function call graphs with an explicit feature map inspired by a linear-time graph kernel.

Christodorescu et al. in [40] proposed an automated technique to mine malicious behavior present in known malware. Execution traces collected from malware and benign samples are used to construct dependence graph and further to mine malicious behavior by differentiating between dependence graphs of malware and benign programs.

In [41], alternative sequences for actions, such as proxying, keystroke logging, data leaking, downloading and executing program are crafted, using data-flow analysis technique. This behavior model is used for detection of malware.

[42] uses call graphs to represent malware samples and extract certain variations, enabling the detection of similar structural similarities between samples. Pairwise graph similarity is calculated by graph matchings, which approximately minimizes the graph edit distance. To discover similar malware samples, the method uses several clustering algorithms, such as k-medoids and Density-Based Spatial Clustering of Applications with Noise (DBSCAN).

To learn the semantics of attack behaviors in malware, Meng et al. [43] uses deterministic symbolic automaton (DSA) based approach for the purpose of android malware detection and classification. It learns DSA by detecting and summarizing semantic clones from malware families and then extracts semantic features from the learned DSA to classify malware according to the attack patterns. In order to improve the understanding of malicious behavior, Narayanan et al. [44] proposed to enrich the feature space of a graph kernel that inherently captures structural information with contextual information. Contextual information provides information about the context under which the sub-structure is reachable during program execution. Narayanan et al. [45] proposes an online machine learning based framework, based on features obtained from inter-procedural control-flow graphs to perform accurate malware detection. Meng et al. [46] captures the common attack features and evasion features to automatically generate new android malware.

Though, graph based behavioral modeling techniques [47, 48] have higher detection rate, they suffer from compute intensive nature of graph mining.

4. **Instruction Trace**

   In this category, we survey the techniques based on the Operational code (opcode), which is the part of the instruction in machine language representation.

   Bilar et al. [49] showed that distribution of opcodes differ significantly in malware and benign programs and hence, rarer opcodes found can be used for prediction of malware.

   Similarly, authors in [50] proposed frequency of opcode sequences to model the malware behavior. To this end, they disassembled the executable to build opcode profile, containing the list of opcode and their frequency of appearance for malware and benign dataset. Further, they used weighted frequency to make feature vector.

   [51] uses opcode sequences in order to construct feature vector representation of the executable and finally train a machine-learning classifier to be used for detection purpose.

   Runwal et al. [52] proposed graph method by first extracting opcodes from both malware and benign samples and calculating their frequencies. A graph of opcode versus frequency is used to differentiate between malware and benign samples.

   The main issue with these techniques is scalability. It requires a significant amount of work to model each program based on instructions. The code size increases day by day, which limits the scalability of this method.

2.4.1 Mobile Devices based Techniques

Current anti-malware tools have not been so successful in defending ever-evolving malware attacks and exploits. Malware solution proposed for the desktop platform have a high performance penalty on modern mobile devices such as tablets,

smartphones. In this section, we survey malware detection techniques proposed for mobile devices such as smartphones and other embedded devices, particularly low-cost solutions.

Zhang et al. [53] proposed to build a cost-efficient way to detect malicious behavior and prevent vulnerability exploits in resource-constrained computing platforms. To this end, their method first tests an application under trusted third party and extracts a behavior model from its execution paths. The user has to download the behavioral model along with the tested application binary. At runtime, the application is monitored against this behavioral model. The behavior model can be further reduced by the publisher through static analysis.

Dinaburg et al. in [54] proposed an external malware analyzer called Ether, which uses hardware virtualization techniques such as Intel VT. It resides completely out of target OS environment and thus, leaves no in-guest software components vulnerable to detection.

[55] proposes microarchitectural execution patterns to detect malware programs by comparing with the execution patterns of the known malware programs. The approach first uses unsupervised machine learning to build profiles of normal program execution based on data from performance counters, and then uses the built profiles to detect significant deviations in program behavior that occur as a result of malware exploitation.

SmartSiren [56] performs virus detection by collecting communication activity from smartphones and performing statistical analysis, to detect single-device and system-wide abnormal behaviors, based on communication data such as usage of SMS/MMS messages. [57] proposes a behavioral detection framework to detect mobile malware by using a trained support vector machine classifier to capture the order of actions of application.
Most of these techniques ignore semantics of program behavior, which paves an easy way for the malware to evade using obfuscation techniques. Table 2.1 summarizes the aforesaid dynamic techniques. It also differentiates hardware and software based approaches.

Table 2.1: Summary of Dynamic Malware Analysis Techniques

Techniques	Ref.	Methods	Hardware (H) or Software (S)
Function Call Monitoring	[27]	windows native API and system calls	S
	[28]	system call without arguments	S
	[29]	discontiguous system call patterns	S
	[30]	automated method of extracting API	S
	[31]	system calls/action with/without arguments	S
	[32]	interaction of benign programs and OS	S
	[36]	sequence of system calls, using n-gram and genetic algorithm	S
Function Parameter Analysis	[33]	scalable clustering approach	S
	[34]	unsupervised detection using system call arguments and sequences	S
	[35]	arguments passed to each system call	S
	[37, 38]	attack behavior model to detect and classify javascript malware	S
Information Flow Tracking	[39]	function call graphs on android	S
	[40]	dependence graph of malware and benign	S
	[41]	patterns of proxying, keystroke logging, data leaking, downloading	S
	[42]	clustering algorithms, such as DBSCAN	S
	[43]	deterministic symbolic automaton (DSA) based approach	S
	[44]	graph kernel with structural and contextual information	S
	[45]	machine learning approach using interprocedural control-flow graphs	S
	[47, 48]	graph based behavior modeling	S
Instruction Trace	[49]	opcodes difference in malware and benign	S
	[50]	frequency of opcode sequences	S
	[51]	opcode sequences and machine learning classifier	S
	[52]	graph method by extracting opcodes	S
Mobile Devices based	[53]	behavioral model extracted by trusted third party	S
	[54]	hardware virtualization techniques such as Intel VT	H
	[55]	microarchitectural execution patterns (performance counters)	S
	[56]	communication activity from smartphones (SMS/MMS)	S
	[57]	order of actions of an application	S

2.5 Exploits

Runtime attacks on memory have been the predominant attack vectors against software programs for more than two decades. Adversaries have exploited memory vulnerabilities, such as buffer overflow to hijack the control flow of the software

programs. The principal reason for the success of these attacks can be attributed to the fact that large portions of software applications are implemented in type-unsafe languages (C, C++ or Objective-C), which lack the bounds checking on data inputs. On top of that, even type-safe languages depend on interpreters (e.g., Java depends on Java virtual machine) that are in turn implemented in type-unsafe languages. As software applications and compilers continue to become more complex, memory errors and vulnerabilities will be inevitable. The common example of memory vulnerability is the stack overflow, where the attacker overflows a local buffer on the stack and overwrites a function's return address [58]. Although current defense mechanisms (e.g., by using stack canaries [59]) protect against this attack strategy, other exploitation techniques (e.g., leveraging heap [60], format string [61], or integer overflow [62] vulnerabilities) exist till date. Exploiting a vulnerability to gain control over application control flow is only the first step of a runtime attack. The next step is to execute malicious programs. Earlier, the attacker used to realize this by injecting the malicious code into the application's address space and then redirecting the control flow to the injected code. However, with the broad deployment of non-executable memory or data execution prevention (DEP) countermeasure, which ensures that the writable page in memory is non-executable, the classical injection attacks are harder to perform.

In response to this, code reuse attacks, such as return-into-libc [63, 64] and return-oriented programming (ROP) [65], emerged as a new attack vector. In a code reuse attack, the attacker does not inject the code but instead use the code already present in memory. The malicious operation is performed by chaining together existing sequences of instructions (called gadgets) that are present in the library or application code. Currently, many exploits use ROP or its variants to bypass the existing defense techniques and transfer the control flow of the program to the malicious payload. Commonly, these payloads are intended to perform arbitrary code execution, privilege escalation, and extraction of sensitive information.

2.6 Code Reuse Exploits

Return-oriented programming (ROP) [65] is the most popular code reuse attack. It is also considered as a generalization of return-to-libc [60] attack, where the attacker hijacks the control flow and causes the program to return to sensitive library functions, e.g., libc. The key idea of ROP attack is to combine short code sequences, called gadgets, that are present in the program's address space (e.g., shared libraries and the executable) to perform arbitrary computation. Similar to other runtime attacks, it exploits memory related vulnerabilities (e.g., stack, heap or integer overflows) in the software during runtime. First, the adversary aims to gain control over the control flow which is followed by execution of malicious payload. ROP exploit has been performed on several platforms including x86 [65], SPARC [66] and ARM [67]. We discuss a brief overview of ROP and JOP exploit below.

In a ROP exploit, the attacker finds the gadgets in program's address space and orchestrates them to be executed in sequence to perform their intended task. Each gadget performs some computation, such as loading a value from memory into a register or adding two registers.

The gadgets are chained together by controlling the target of a gadget's indirect branch (jump or return) to transfer execution to the beginning of the next gadget in the sequence. In a general ROP attack, gadgets end with the `ret` instruction and the attacker chains gadgets by writing appropriate values over the stack. Figure 2.1 depicts a typical memory layout for a ROP exploit. Along with the return addresses, the adversary also writes several data words in memory that are used by code sequences, such as "pop eax" instruction in gadget 1 use DATA WORD 1. First, the stack pointer points to the first return address of the payload, which returns to gadget 1. After the execution of gadget 1, its return instruction "ret" advances the SP by one memory word, loads the next return address (RET ADDR 2), and finally transfers the control flow to the next gadget (gadget 2). The stack pointer acts a virtual program counter to execute the gadgets in sequence. The combined execution of gadgets results in malicious operations. The gadgets

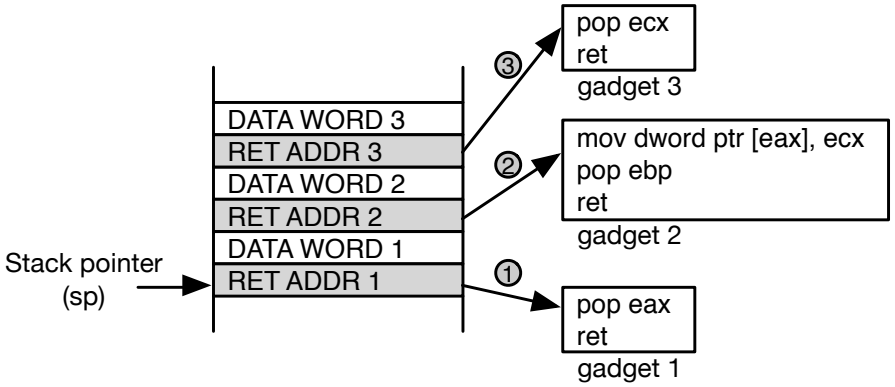


FIGURE 2.1: Program Memory Layout of a ROP Exploit

are identified by using static analysis on program binary and its linked shared libraries. Moreover, these gadgets are Turing-complete, i.e., they can be used to perform any arbitrary computation.

The ROP gadgets are also formed using unintended instruction sequences. Since x86 is a variable-length instruction set architecture varying from 1 to 15 bytes of instructions, the instructions can also be formed by pointing the program counter to the middle of instruction. Such instructions are not known during compile time. Thus, the attackers get larger search space, where they can find the gadget sequence, when starting at an offset that was not intended to be the beginning of an instruction.

Jump-oriented programming (JOP) is another class of code reuse attack that uses indirect jumps and calls rather than return instructions [68, 69]. In general, JOP uses the combination of `pop-jmp` to emulate return instructions. It does not require the stack pointer as a base register to reference code pointers, the attacker instead uses an *update-load-branch* sequence with general purpose registers [69]. Similar to ROP and JOP, **call-oriented programming (COP)** is used to refer to ROP variant techniques that employ indirect calls [70, 71].

2.7 Exploits Defensive Methods

Code reuse attacks have been acknowledged as a powerful exploitation technique that can bypass the existing defense mechanisms. In response, there has been much effort to build the defense techniques to protect against these attacks. In this section, we overview the defense approaches into following categories:

1. **Control Flow Integrity Approaches**

   Abadi et al. introduced control flow integrity (CFI) [72] as an approach to preventing runtime attacks by restricting the control flow transfers (e.g., jump, call, and return instructions) to follow the statically-determined control flow graph (CFG) of the program. This approach can prevent code reuse attacks such as ROP, as they cause the program to deviate from normal control flow graph. Most CFI approaches follow a two-phase process. In the analysis phase, they obtain the CFG of the program which approximates the set of the legitimate control flow transfers. This CFG is then used to at runtime to ensure that at all control flow transfers follow the CFG. During the analysis phase, the CFG is obtained either from the source code or binary of the program. However, due to the limitation of the static program analysis, precise CFG of the program cannot be obtained. Therefore, many defense techniques instead enforce a coarse-grained policy, e.g., returns must be call-preceded, and indirect calls point to the beginning of functions [73–75]. These CFI implementations are weaker and can be defeated by ROP attacks [70, 71]. Many CFI implementations also use a shadow stack [76–78], to store the return address in a safe location upon each function call. It is used to verify the return address during execution of return instruction. This largely reduces the gadget space available for performing ROP exploits.

   Although CFI is a promising approach, several factors hinder the deployment of CFI in practice. First, obtaining a precise CFG is a difficult task. The reasons are: 1) source code is not available for all the applications, thus limiting the compiler-based techniques; 2) binaries generally lack relocation or debug information; and 3) the performance overhead is high due to code

rewriting and runtime checks [79]. Many CFI implementations focus on addressing these limitations [79].

2. **Randomization-based Approaches**

   Address Space Layout Randomization (ASLR) is an OS mitigation technique that randomizes the segments of a program (including the text segment, the stack and base addresses of most of the libraries and executables) around in memory, to prevent the attackers from predicting the address of useful gadgets. However, the attackers are still successful in performing ROP attacks because of two reasons. First, there exist many non-ASLR enabled modules that are used by the attackers to perform ROP attack. Second, an attacker may use just-in-time code reuse attack [80] that compiles ROP on the fly to bypass ASLR.

3. **Runtime Defenses**

   Several other runtime defenses aim to protect against ROP exploits and its variants. DROP [81] builds its defense mechanism against traditional ROP exploits based on the observation that these exploits contain a long consecutive sequence of return instructions, and the gadgets are small in length. However, the approach can be defeated using ROP exploits with long gadgets [82]. ROPDefender [83] instruments call and return instructions and maintains a shadow stack to verify that the return address is present in the shadow stack. ROPGuard [84] uses heuristics to detect ROP exploits, such as instruction preceding the return address must be call instruction. The main drawback of this approach is that it performs the validation only when a critical Windows OS function is called. This makes it vulnerable to ROP attacks that can jump over these checks (e.g., using hook hopping as discussed in *C1* in Section 6.2.3) and those ROP attacks that do not use any critical Windows function [82].

4. **Recompilation-based Defenses**

   Some defenses aim to remove useful ROP gadgets from the compiled binary

Table 2.2: Summary of Exploit Defense Techniques

Techniques	Software	Hardware
Control flow integrity	Abadi et al. [72]	Cfimon [75], BR [76]
	Zhang et al. [73]	Davi et al. [77]
	Bletsch et al. [74]	Das et al. [78]
Randomization-based	ASLR	
Runtime defenses	DROP [81]	
	ROPDefender [83]	
	ROPGuard [84]	
Recompilation-based	G-Free [85]	
	Davi et al. [86]	

using recompilation technique. G-Free [85] removes unintended return instructions and encrypts return addresses so that ret-gadgets become harder to use. Similarly, the return-less kernel [86] entirely removes the `c3` byte (the opcode of `ret`) from all instructions and replaces valid returns with a lookup into a table containing the valid return sites.

Table 2.2 summarizes the hardware and software based exploit defenses.

2.8 Summary

In this chapter, we discussed the background and related work on malware. We highlighted the different techniques that are used for malware analysis and detection. We presented an overview of code reuse attacks, particularly ROP exploit and its variants. Finally, we discussed the defense mechanisms that are proposed to defeat code reuse attacks.

3

Online Malware Detection Using Machine Learning Approach

3.1 Introduction

In recent years, malicious software (in short "malware") has skyrocketed in the computing platforms, such as smart phones and tablets. Recent McAfee report [87] shows that mobile malware samples grew by 16% during third quarter of 2014 with total samples exceeding 5 million and by 112% in 2014. However, effective malware detection has been a challenging task because sophisticated techniques are used by the malware writers to exploit the system vulnerabilities.

Despite the fact that significant amount of work has been done in malware detection, they have not been successful in combating the ever-evolving and the sophisticated malware. Typical static solutions — such as antivirus, scanners and anti-malware tools — use a signature based method to detect the malware [12]. Unfortunately, adversaries use obfuscation techniques (e.g., code encryption) [15] and write several variants of the same malware to evade the signature based static detection techniques. In response to this, dynamic approaches were proposed, which analyze the program behavior during execution. Principal dynamic techniques include virtual machine inspection [88], function call monitoring [27, 28], dynamic binary instrumentation [89] and information flow tracking [90]. Though these approaches can potentially detect the obfuscated malware and also the malware variants, they require a large amount of resources and have a substantial overhead on the system. These difficulties often limit the malware detection to use a static approach (e.g., antivirus, scanners), however, they can be easily evaded due to the known limitations.

System calls based techniques have been recognized as a promising dynamic approach for malware detection. System call patterns provide effective information about the runtime activities of a program, which can be used to characterize the malicious behavior. Recently, researchers explored clustering techniques to group the system calls of a sample to identify the malicious patterns. To this end, [91] proposed n-gram approach to cluster the system calls, whereas other methods grouped them based on common OS resources [33, 92]. However, the current implementations [33, 91, 92] are mainly software-based, which have the same level of vulnerability as the software. Currently, adversaries use advanced techniques (e.g., anti-virtual machine inspection and anti-debugger [16]) against these protections to hide the malicious activity and finally evade the detection. To give an instance, a malware [13] uses anti-debugger techniques — anti-ptrace, anti-sigtrap and anti-breakpoint — to detect the presence of malware detector in the host and exits cleanly once it finds them. Most importantly, software approaches are infeasible for certain classes of malware; such as rootkits and bootkits achieve their malicious intents by infecting the kernel [14]. Since the rootkits have the similar privilege as

the OS (Ring 0), they can disable and evade the software protection. In addition, the software methods are resource demanding, which causes a high energy demand and a substantial performance overhead, when employed on resource-constrained systems.

In this chapter, we present the first system call based approach using hardware-enhanced architecture that employs machine learning technique for malware detection. We name it GuardOL (guard online). GuardOL uses a novel frequency-centralized model (FCM) for feature construction to learn the malicious behavioral patterns from known malware samples. Our frequency-centralized model takes the frequency of resource-critical system calls into account and constructs features by grouping system calls using comprehensive rules to capture the semantics of malicious behavior. To this end, GuardOL extracts the system calls (with their arguments and return values) during execution on the processor and groups relevant system calls to construct features using FCM. The features obtained from the malware and benign samples are used to train multilayer perceptron (MLP), an artificial neural network model, which is used at runtime to perform the classification of the running program as malware or benign. We develop an architectural design of GuardOL based on our proposed methodology. For the proof of concept, we implement our model in FPGA. We leverage the advantages of FPGA platform to obtain a high performance training and detection at a low cost and reconfigurability for post-fabrication functionality upgrade to adapt to new malware samples. Reconfigurability of FPGA also allows sharing of hardware for classifier training and runtime detection.

Our approach aims to capture the semantics of malicious behavior using the proposed frequency centralized model and therefore has the potential to detect malware variants and even zero-day (previously unseen) attacks. Compared to the software approaches, GuardOL is resistant to aforesaid advanced malware techniques, as it is based on hardware. Unlike software techniques, it cannot be disabled or discovered by sophisticated techniques (e.g., anti-virtual machine and anti-debugger) and thus malicious activities will not be hidden from detection. Moreover, our implementation offers a power-efficient malware detection design,

which has low resource demands and negligible performance overhead on the system. The FPGA implementation of GuardOL consumes 0.36 W during training and 0.264 W at runtime. Our results show that GuardOL achieves faster detection with zero performance penalty on the processor.

Our method supports an early prediction of malware samples as it performs the detection during their execution. The early prediction facilitates the real-time capability to the malware detection. In general, malware needs to pass through several phases (such as debugging, environment setup) before they can perform malicious operations in the system. Hence, it is an additional advantage if the malware samples can be detected in their initial stages, before the major damages have been performed in the system. The experimental results on our dataset show that GuardOL can detect 47% of malware samples within first 30% of their execution, while 98% of the samples after their execution, with <3% false positives.

Our contributions are:

- We propose a frequency-centralized model (FCM) for feature construction, which uses comprehensive rules to group the system calls using their arguments and return values, to capture the semantics of malicious behavior.
- We present an architectural design of GuardOL based on the proposed FCM and machine learning classifier (MLP) to detect malware at runtime. We explore the design considerations for GuardOL implementation.
- We explore the early prediction of malware and show that it has the potential to detect malware in its early phase of execution.

GuardOL employs system calls and their arguments to model the behavior of the program, thus it can detect malicious attacks that leave evidence in the system calls. In general, most of the malware invoke system calls to perform malicious activities. Kernel-level malware that does not modify the system call library can be detected by our method. But, some kernel rootkits (e.g., SucKIT, Adore-ng, Sk2rc2) that manipulate the instructions inside system calls or the entries of the

system call table [93] cannot be detected by our method. Nevertheless, GuardOL is an extensible architecture that can support instruction-level evidence to detect such rootkits. We reserve this extension for future work. In this article, we assume OS and typically the system call libraries must be trusted.

The rest of the chapter is organized as follows: In Section 3.2, we discuss the malware preliminaries. We review the related work in Section 3.3. We present our feature construction model in Section 3.4 and the offline evaluation in Section 3.5. Section 3.6 describes our proposed hardware design of GuardOL. In Section 3.7, we present the hardware implementation results. and finally conclude in Section 3.8.

3.2 Malware Preliminaries

Since the early days of computing, malware has evolved from the simple exploits into the complex ones in the forms of — virus, worm, trojan, adware, spyware, backdoor, flooder, botnet, rootkit and bootkits [12]. Over the years, the motivation of malware authors has changed from exploits-for-fun to money-making business. With the arms race between security professionals and malware writers, the present day malware has highly evolved, which uses sophisticated techniques to exploit the vulnerabilities. Malware uses several channels to penetrate in the system, e.g., phising emails, usb, memory card, corrupt downloaded files, click on links, repackaged into normal applications, browser utilities, abusing application stores or exploiting old vulnerabilities [94]. Our work is based on the behavior of the malware in the host system, i.e., after they have penetrated in the system.

3.2.1 Dataset

Our dataset contains 472 Linux malware samples collected from Virusshare [95] and VX Heaven [96], which includes trojans, exploits, viruses, worms and rootkits. We used Virustotal [97] online tool for the classification of malware samples and identification of their types, presented in Table 3.1. We selected 371 benign

Table 3.1: List of malware families in our dataset

Types	Family	#Sample
Backdoor	Agent, BO, Bodoor, Boost, Dancer, Divine, Explodor, Fpath, Hydgo, IrcShell, Iroffer, Lala, NetBus, Phobi, Rooter, SitC, Small, SpyEye, SSh, Suffer, Unfstealth	39
Exploit	Acpi, Apache, Bind, Brk, Da2, Epoll, FormatStr, Freeciv, Glc, Ipb, Kmod, Linux, Local, Mysql, Named, Old, OpenSSL, Race, Rpc, ShellCode, Small, SSHD22, Ssl, Vmsplice, WuFtpd, Xpl	114
Flooder	Slice, Small	3
HackTool	Masan, Scanap, Sh, Sshbru, BF, CleanLog, Small, SVScan, Usmel, Vcmer	25
Net-Worm	Kork, Mworm, Ramen, Coptic, Hijack, Lion, Old, Slapper, Sorso, Usmel	20
Rootkit	Agent, Gabitzu, Matrics, R3dstorm	56
Trojan	Generic, Hopbot, Regen2k, Small, Agent, Blitz, Hacktop, Logftp, Ris, Zapchast	16
Virus	Alaeda, Bi, Binom, Caveat, Clifax, Diesel, EthClean, FortyTwo, Grip, Joper, Little, Mandragore, Nuxbee, Orig, Osf, Ovets, Piltot, Quasi, Rike, RST, Satyr, Sickabs, Siilov, Silvio, Small, Snoopy, Spork, Svat, Telf, Thebe, Thou, Winter, Wowood, Xone, Brundle, Dido, Eriz, Grip, Impok, Little, Mais, Mandragore, Osf, Pelf, Piltot, RcrGood, RST, Satyr, Small, Svat, Telf, Thebe, Vit	199
		472

applications on Ubuntu OS 12.04, consisting of programs from several categories — internet, games, system tools, office, sound, video, multimedia and utilities. We divided the dataset into 70%-30%. 70% of the samples (i.e., set-A) were used for classifier training and testing using a standard 10-cross validation approach. Remaining 30% of the samples (i.e., set-B) were used for the evaluation of early prediction.

To generate system call traces, all malware samples were executed in a virtualized environment, consisting of 32-bit Ubuntu (12.04) OS. We extracted system calls and parameters using *strace* (a Linux debugger to trace system calls and signals). In a similar way, benign samples were executed using *strace* to extract the system calls in a normal environment. Common user operations were performed on the benign programs in order to trace their systems calls. We used WEKA tool [98] for the offline evaluation of machine learning classifier.

3.2.2 Low-level Malware Behavior Observed in Our Dataset

Malware performs a series of actions in order to accomplish their malicious intents, for which they need to use OS resources such as filesystem, memory, process and network. System calls are invoked to access these OS resources. They are commonly used in malware analysis to abstract the malicious behavior, as they accurately describe the runtime activities of the program [28, 92].

We studied the behavior of malware samples to observe the patterns of system calls occurred during their execution. We consider only security-critical system calls, which are those system calls that modify the state of OS resources [33]. In this work, first we listed the most frequent system calls that modify the state of OS resources from malware programs. As our approach is based on determining the common system call patterns found in malware, we neglected those system calls that appear only in few samples. Finally, we selected 64 most frequent system calls.

Table 3.2: Low-level actions observed in our malware dataset

Resource	Low-level behavior	System calls
Filesystem	Create/read/write/delete/copy	`read`, `write`, `creat`, `open`, `openat`, `unlink`
	Search executable/library files	`opendir`, `chdir`, `access`, `readdir`
	Modify file/properties	`utime`, `chmod`, `ftruncate`, `rename`
	Get file/directory information	`getdents`, `fstat`, `fstat64`, `fadvise64`
	Execute a file	`execve`
Process	Change signal actions	`rt_sigaction`, `rt_sigprocmask`
	Kill process	`kill`, `tgkill`
	Switch process context	`sched_yield`
Network	Receive/send information	`send`, `bind`, `connect`, `recvfrom`
	Monitor network events	`poll`, `epoll_create`, `select`,
	Receive command from network	`recvfrom`, `ioctl`
Memory	Increase data memory	`brk`
	Map file/device to memory	`mmap`, `mmap2`, `munmap`, `old_mmap`
	Set memory protection	`mprotect`

Table 3.2 lists some common actions performed by malware samples on Linux OS, along with the corresponding security-critical system calls. We observe that the actions listed in the table are common to the benign programs. However, the combination of these actions may lead to the stealthy operations by malware. To give a few examples: in order to replicate itself, a virus first searches the executables in the system, copies its content into them and finally executes those infected executables; some exploits first change the signal action (e.g., interrupt, keyboard input) to redirect the execution to their own code upon some signal action. And then they perform malicious activities, which cannot be recognized by the OS, as the signal action has been modified. Therefore, in our malware detection approach (as explained in Section 3.4) we aim to learn the semantics of the high-level malicious behaviors, which use low-level actions to perform their intended goal.

3.3 Related Work

In this section, first we overview the common software-based malware detection approaches. Second, we discuss the detection techniques that utilize hardware features. Third, we summarize the general hardware-based approaches that help to protect from the exploitation of vulnerabilities.

3.3.1 Software-based Malware Detection

A considerable number of works have been done in malware detection. Malware detection techniques can be broadly classified into two categories — Static and Dynamic. Static approaches (e.g., antivirus, scanners) analyze a structure of the program by inspection, without executing the program. They use the signatures of the known malware samples. But these techniques can be easily evaded by the simple program transformation or code obfuscation (e.g., polymorphic malware) [99]. Malware authors write several malware variants, which have similar functionality but different signatures, to evade static protections. Besides, it requires a lot of manpower and time to extract the signatures of each malware. On a contrary, dynamic techniques analyze the program behavior during execution. The principal techniques include function call monitoring [27–29, 33, 38, 91, 100], virtual machine introspection [54], information flow tracking [39, 40], instruction trace monitoring [49–51]. Since they monitor the program behavior during the execution, they can potentially detect malware variants as well as the obfuscated malware. However, most of the previously proposed techniques have a high false positive rate, a substantial performance overhead and high resource demands. In addition, recent malware employs sophisticated concealment strategies (such as anti-debugging, anti-virtual machine) to hide its stealthy operations and evade the protection [16].

System call patterns provide effective information about the runtime activities of the program. In the past, many system call based techniques have been proposed.

Forrest et al. [28] first proposed anomaly detection using small sequences of system calls. Maggi et al. [34] used system call arguments and sequences to capture interrelations among different arguments of a system call, which were utilized to study time correlations to detect abnormal behaviors. Canali et al. [91] explored the data-mining techniques (such as n-gram, m-bag, k-tuples) to group the system calls and used machine learning techniques to capture the malicious behavior. However, these data-mining techniques generate large number of features. As a result, they have high resource demands and a substantial performance overhead. In response to this, Chandramohan et al. [33] proposed a scalable clustering approach, BOFM, to group the system calls based on the common OS resource identifier. Though BOFM significantly reduces the feature size, it does not consider the frequency of system calls for feature construction. This can be exploited by malware to perform attacks (such as denial of service and CPU starvation) by executing system calls repetitively. In contrast to this work, we adopt a frequency-centralized feature construction model to group the system calls and capture malware behavior based on more comprehensive rules (as described in Section 3.4). Compared to BOFM, our method significantly reduces the false positive rate. Furthermore, all these works consider offline analysis of malware, whereas our work focuses on online malware detection.

Software-based solutions have known limitations. They offer the same level of vulnerability as the software and thus, they can be bypassed by using sophisticated techniques [13, 16]. Moreover, certain class of malware (e.g., kernel-mode rootkits) can easily evade them, and thus making them undesirable for advanced malware detection. In addition, these solutions are resource demanding causing high energy consumption and a substantial performance overhead, when employed on resource-constrained systems.

3.3.2 Hardware-based Malware Detection

A number of previous works leverage architectural features for malware analysis and detection. Bilar et al. [49] used the difference of opcodes between known

malware and benign programs for malware prediction. Similarly, other methods employ frequency of opcodes [50] and sequences of opcodes [51] to model the malicious behavior. Runwal et al. [52] proposed a graphical technique to find the similarity of the opcode sequence. However, these techniques require significant amount of work to model each program based on instructions. As the code size increases day by day, modeling program based on opcodes becomes a time-consuming process. Moreover, with the increment in code size, the memory requirement also increases. This will also result in significant performance overhead on the system, as each instruction of the program has to be traced.

Demme et al. [101] proposed the use of hardware performance counter to monitor the lower level micro-architectural parameters such as instruction per cycle, cache miss rate. Tang et al. [102] built baseline models of benign program execution using unsupervised machine learning to detect the deviations that occur as a result of malware exploitation. NumChecker [93] detects malicious modifications to a kernel function (system call) by checking the hardware events including total instructions, branches, returns and floating-point operations. Following the similar approach as in [101], Ozsoy et al. [103] proposed the design of a malware-aware processor using architectural events (e.g., frequency of memory read/writes, immediate branches taken, frequency of opcodes) as the features and machine learning for the classification of malware. The authors in [104] applied singular value decomposition technique to reduce the feature size. All the above methods use low-level hardware features to model the malicious behavior. They have low performance overhead, however, they are limited by a high false positive rate (about 10% [101, 103]). In comparison, our technique uses high-level features (i.e., system calls and arguments) to model the program behavior and has lower false positive rate, which is more significant in malware analysis.

Rahmatian et al. [105] proposed host-based intrusion detection using FPGA, which uses system call sequences to characterize the correct system behavior. The approach of system calls extraction from the processor is similar to ours (i.e., hardware modification to trace system calls using trap instruction), however, they use

finite state machine (FSM) of the program generated offline to do runtime monitoring. The main limitations of this approach are: each program needs to be assisted by system calls based FSM; it does not allow unknown programs to run; and FSM of system calls would be significantly large for complex programs, thus demanding large memory size. In contrast, GuardOL uses semantics of malicious attacks and machine learning approach, which results in small feature models and applicability to unknown programs.

3.3.3 Hardware-based Protection Approaches

Malware also exploits the system vulnerabilities by performing attacks such as buffer overflow and code reuse. A number of hardware approaches aim to protect from these attacks. Consequently, these hardware techniques also help to protect from the malicious operations. In this section, we brief the relevant hardware approaches that make it harder for malware to exploit the vulnerabilities.

Buffer overflow remains the most prevalent exploit till date. In the past, several architectural techniques [106–109] have been proposed to prevent from such attacks. With the introduction of DEP or W$\oplus$X [110, 111] protection, which allows memory page to be either writable or readable but not both at the same time, the traditional code injection attacks can be prevented. However, the adversaries developed a new type of attacks, code reuse attacks (CRAs), which use the existing code sequences instead of code injection. Return Oriented programming (ROP) [65] and Jump Oriented Programming (JOP) [68] are two common types of CRAs. Effective solutions based on control flow integrity [75, 112–114] have been proposed to defend against CRAs. A complimentary line of research includes architectural mechanisms that aim to secure embedded systems by verifying the integrity of the code [115].

```
1.  openat(AT_FDCWD,"/usr/bin",..)  = 2
2.  brk(0) = 0x9ec8000
3.  brk(0x9ef1000) = 0x9ef1000
4.  getdents(2,..,32768) = 32536
5.  open("file1.out",O_RDONLY|O_CLOEXEC) = 3
6.  fstat64(3,..)  = 0
7.  read(3,.., 1024) = 0
8.  open("../libc.so.6",O_RDONLY|O_CLOEXEC) = 4
9.  fstat64(4,..)  = 0
10. read(4,.., 1024) = 0
11. mmap2(..,4,0) = 0xb7619000
12. open("VirusShare_4da7b",O_RDONLY) = 13
13. open("/tmp/temp0",O_RDWR|O_TRUNC,01) = 14
14. read(13,..,8000) = 6633
15. write(14,..,6633) = 6633
16. chmod("/tmp/temp0",0100770) = 0
17. utime("/tmp/temp0",..)  = 0
18. execve("/tmp/temp0",..)  = 0
```

FIGURE 3.1: Sample malware system call trace

3.4 Feature Construction

Feature selection is a crucial step of any machine learning based approach. The performance of any machine learning classifier depends heavily on how closely the features represent the characteristics of the class. In this section, first we brief the two approaches — n-gram [91] and BOFM [33] — that have been used by previous approaches for feature construction of malware. Then we propose our frequency-centralized model (FCM).

TABLE 3.3: Features for malware trace (Fig. 3.1) using different techniques

4-grams	
1. `openat, brk, brk, getdents`	9. `fstat64, read, mmap2, open`
2. `brk, brk, getdents, open`	10. `read, mmap2, open, open`
3. `brk, getdents, open, fstat64`	11. `mmap2,open,open,read`
4. `getdents, open, fstat64, read`	12. `open,open,read,write`
5. `open, fstat64, read, open`	13. `open,read,write,chmod`
6. `fstat64, read, open, fstat64`	14. `read,write,chmod,utime`
7. `read, open, fstat64, read`	15. `write,chmod,utime,execve`
8. `open, fstat64, read, mmap2`	
BOFM	
1. `openat, getdents`	4. `open, fstat64, read, mmap2`
2. `brk`	5. `open, read`
3. `open, fstat64, read`	6. `open, write, chmod, utime, execve`
FCM	
1. `openat, getdents`	4. `mmap2`
2. `brk`	5. `open, read, write, chmod, utime, execve`
3. `open, fstat64, read`	

3.4.1 *n*-gram Technique

An n-gram is a popular data mining technique used in feature construction [91]. This approach groups "n" system calls that appear in a consecutive order to form a feature. We illustrate this by an example (Fig. 3.1 is used throughout the chapter to illustrate the feature construction). For system call trace shown in Fig. 3.1, as listed in Table 3.3, there are 15 features obtained for 4-grams (n-gram technique, where n=4).

We performed feature construction using 4-grams and 6-grams techniques, which have better performance as explored in [36, 91]. The number of features using 4-grams and 6-grams techniques are 22134 and 56908 respectively, as shown in Table 3.4. Since the feature size is quite large, it requires substantially large

memory for storing features and also involves heavy computations during training and runtime detection. Consequently, the high performance overhead limits the practical use of this approach.

3.4.2 BOFM

BOFM [33] is a scalable framework, which clusters system calls based on actions on common OS resources. To this end, BOFM uses 3 rules: the same set of actions on OS resource instance is considered as one feature; the sequence of actions is not considered; and identical action sets performed on two OS resource instances are taken as a single feature. Table 3.3 lists the features obtained for the trace in Fig. 3.1.

Using this approach on our dataset, the total feature size reduces to 258 (as shown in Table 3.4), which is a significant achievement. However, the approach does not consider the frequency of system calls, which can be leveraged by malware. Malware may consume resources of system by executing system calls repetitively, e.g., `brk()` system call is executed repetitively by *Brk.c* exploit. Such a class of malware can lead to the denial of service and CPU starvation attacks in the system.

3.4.3 Frequency-Centralized Model (FCM)

In this work, we propose a novel frequency-centralized feature construction model (FCM) to capture the semantics of malware behavior. We consider the frequency of resource-critical system calls for feature construction in order to detect malicious attacks that execute repetitive system calls to overload the system. Our proposed model is based on the high-level semantics of malicious behavior. We adopt a dynamic approach to monitor security-critical system calls along with their arguments and return values to identify malicious combinations through machine learning. We aim at a generic representation of a malicious behavior using system calls in order to detect the variants of the malware, and even zero-day attacks that

have similar semantics. To this end, we propose the following rules to capture the high-level semantics of malicious attacks.

Rule 1: Group system calls in sets

The intuition behind this rule is that malicious objectives are accomplished by performing a series of actions on a particular resource. For example, in the above trace, the virus propagates itself by infecting the temporary file. First, it copies its content into the `temp0` file, changes the permission and the accessed time and finally executes the modified file. All these actions performed on a common file reveal malicious intents of the sample. Several variations can be found among viruses to achieve this replication behavior — the required file may be searched in the current/root/parent directories, the target file may be an existing executable, source code, library or temporary file.

We do not consider the *execution order of system calls* in this chapter. Malware may perform its actions by calling system calls in a different order, e.g., the accessed time of `temp0` file may be modified first and the permission can be changed later or the same action can be performed in the reverse order, achieving the desired goal. Given these observations, set structure is an effective way to group the system calls for identifying malware, and in general to discover the malware variants. System calls are grouped in sets based on the common OS resource identifier. An OS resource identifier refers to an instance of the OS resource, e.g., a filename is an instance of filesystem resource. In Fig. 3.1, the directory "/usr/bin" is first opened (using `openat` at line 1) and its information is obtained (using `getdents` at line 4). Hence, `openat` and `getdents` are grouped in a set, as shown in Table 3.3. Similarly, `open` (in line 5), `fstat64` (in line 6) and `read` (in line 7) are grouped into a set based on their action on a common file (i.e., "file1.out").

Rule 2: Trace the memory mapping of file/device

Once the file/device is mapped into the memory (as shown in trace, using `mmap2`), it can be read and written by accessing the user space memory, which does not require the calling of system calls. The detection method based on system calls hence cannot notice the underlying operations on memory-mapped file/device by

the malware. In order to detect such behavior, we consider the mapping of file/device into memory as a separate feature. In our example trace (Fig. 3.1), the file "libc.so.6" is mapped to a user space memory (using `mmap2` at line 11), hence, we take `mmap2` as a feature, shown in Table 3.3.

Rule 3: Include the source file of the write operations

Copying operations involve data propagation, which is a commonly observed behavior of malware. A virus propagates by replicating itself, for which it copies its content into the executable/temporary file partially or wholly, as shown by the trace in Fig. 3.1. Malware may send out the secret information from the host file system, which may lead to the leakage of private information. In order to capture such behavior, the source of the write operations (in Fig. 3.1, the source of the write operation is a virus file) needs to be mapped to the destination file. In the above trace (Fig. 3.1), lines 12-18 reflect the data movement from a virus source file (i.e., "VirusShare_4da7b") into a temporary file (i.e., "tmp/temp0"). In order to capture this behavior, the system calls related to the source and destination files are combined into a set (shown in Table 3.3).

Rule 4: Count the frequency of resource-critical system calls

Malware often executes resource-critical system calls repetitively in order to overload the system resources (such as RAM, CPU), which may result into the crashing of OS, degrading the processor performance and draining the battery. And this may consequently lead to the attacks such as denial of service and CPU-starvation in resource limited systems. For example, *Brk.c* exploit uses `brk` system call repetitively to increase the data segment of user program and overloads the main memory. Other exploits execute `rt_sigaction` system call repetitively to change the default signal action for all the signals. Once the default action for the signal is modified, they perform malicious actions, which go unnoticed by the OS. Therefore, recording the frequency of these system calls is important to identify similar malicious behavior. In our example trace (Fig. 3.1), the frequency of `brk` system call is taken into consideration during feature construction.

In comparison to the software based solutions that use system calls directly [36, 91] or group system calls executed on common OS resources [33], we adopt a comprehensive semantics-based approach to extract malware behavior. Here we advocate that the above rules are proposed based on the ground facts of malicious behavior (which was observed in our dataset as mentioned in Section 3.2) with justifications as explained above. Our method adopts one rule similar to BOFM, i.e., Rule 1. Because of this, some features in Table 3.3 are common between FCM and BOFM. However, our additional Rules 2-4 are novel and help to improve the robustness of extracted malware features. Rule 2 helps to solve the challenge posed by memory-mapped files/devices to the system call based approaches, which may fail to detect memory operations. Similarly, Rule 3 assists in capturing the data propagation, which is a typical malware behavior. As opposed to BOFM [33], our method takes frequency of system calls into account (Rule 4). This is significant to detect malware samples (e.g., Flooder, Botnet), particularly in resource-constrained systems, that may overload the system resources and end up performing attacks such as denial of service. For instance, a malware can execute resource-critical system calls repetitively to consume available computational resources (e.g., network bandwidth, disk space and processor time), causing resource starvation and crashing the OS.

These rules not only provide robustness to the malware detection but also help to identify typical malicious behavior, which is supported by our high detection accuracy and reduced false positives, as shown by the results in Section 3.5. In addition to this, our approach also reduces the feature size, as compared to the aforesaid techniques (as shown in Table 3.4). In the next section, we will also compare our classifier results with the n-gram and BOFM approaches.

3.5 Offline Evaluation and Analysis

In this section, we present an offline analysis of our proposed feature selection method using software implementation. We also compare our method with the

Feature	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}	n_{11}	n_{12}	...	n_N
Malware #	0	1	1	1	1	1	1	1	1	1	1	1	0	...	0

(a) 4-grams

Feature	b_0	b_1	b_2	b_3	b_4	b_5	b_6	...	b_N
Malware #	0	1	1	1	1	1	0	...	0

(b) BOFM

Feature	f_0	f_1	f_2	f_3	f_4	f_5	...	f_N
Malware #	0	1	2	1	1	1	...	0

(c) FCM

FIGURE 3.2: Feature vectors using different techniques

aforesaid feature construction models — n-gram and BOFM — for our dataset. The features are constructed using the complete trace of system calls obtained after the execution of the sample. In Section 3.5.3, we further evaluate the early prediction using our FCM approach. The features for early prediction are constructed using partial system call traces.

3.5.1 Evaluation of FCM

Our method uses comprehensive rules proposed in Section 3.4 to construct features. We utilize these features to construct a feature vector of size N for each sample, where N is the total number of unique features extracted from both malware and benign samples. The feature vectors are used to train the machine learning classifier. A feature vector represents the presence/absence (represented by 1/0) or frequency of a particular feature in the sample. The frequency of a feature in the sample is considered for the resource-critical system calls as explained by Rule 4. For the system call trace in Fig. 3.1, five features are extracted: f_1: {`openat`, `getdents`}, f_2: {`brk`}, f_3: {`open`, `fstat64`, `read`}, f_4: {`mmap2`}, f_5: {`open`, `read`, `write`, `chmod`, `utime`, `execve`} (shown in Table 3.3). They are embedded in a feature vector as shown in Fig. 3.2(c). Here, we can see that features f_1, f_3, f_4 and f_5 appear only once and f_2 appears twice, while rest of the features are assigned 0.

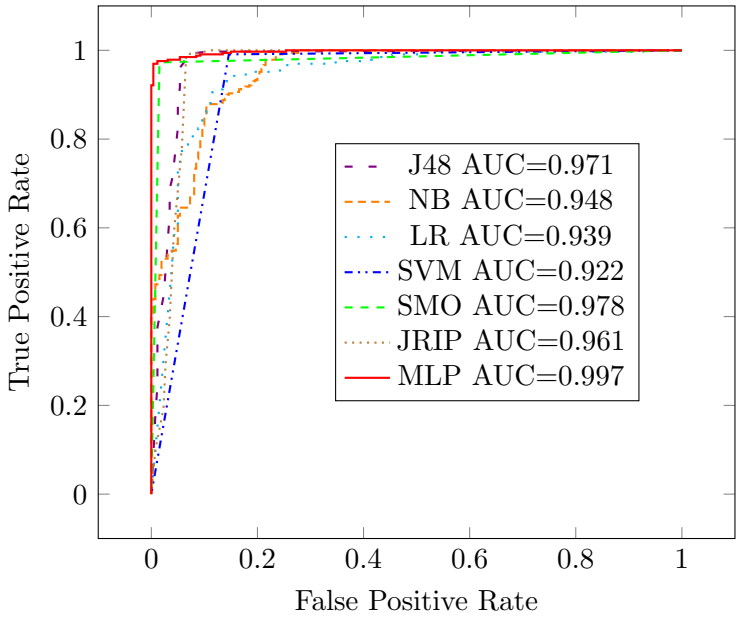


FIGURE 3.3: Performance of classifiers using FCM

We use the feature vectors obtained from malware and benign samples to train the machine learning classifiers and to test the samples. Results are obtained using a standard 10-fold cross-validation on Weka tool. To measure the performance of the classifiers, we plot a receiver operating characteristic (ROC) curve for each classifier. An ROC curve is a graphical plot of a true positive rate (TPR) against a false positive rate (FPR) at various thresholds. TPR is the proportion of malware that is correctly predicted as malware and calculated as TPR = TP/(TP+FN); whereas FPR is the proportion of benign that is mis-predicted as malware and calculated as FPR = FP/(FP+TN). In order to select the best classifier for our dataset, we use the area under curve (AUC) measure for the ROC curves. AUC measures the effectiveness of the classifier to differentiate between two classes, i.e., malware and benign programs in our case. The higher value of AUC has a better classification accuracy. We used the following machine learning classifiers in our experiments: C4.5 decision tree (J48), naive bayes (NB), logistic regression (LR), support vector machine (SVM) using LibSVM, sequential minimal optimization (SMO), RIPPER rule learner (JRIP) and multilayer perceptron (MLP) [98]. Fig. 3.3 depicts the performance of the classifiers. Our results show that multilayer perceptron (MLP) has a higher AUC (0.997) as compared to the other machine learning techniques.

Table 3.4: Comparison of n-gram, BOFM and FCM

	#Features			Performance		
Techniques	malware	benign	total	TPR	FPR	AUC
4-grams	5591	17433	22134	1	0.408	0.796
6-grams	9885	47776	56908	1	0.538	0.731
BOFM	132	219	258	1	0.192	0.904
FCM	119	140	186	0.976	0.012	0.997

With MLP as the classifier, our method has a TPR of 97.6% and an FPR of 1.2% for our dataset. Since MLP has a better classification performance, we choose MLP as a classifier in our method. Another advantage of using MLP is that it is modular and highly parallel algorithm, thus, we can leverage FPGA to perform several levels of parallel computations [116].

Our approach is based on the system call patterns obtained from malware and benign programs. Similar pattern of system calls executed by malware and benign programs will eventually lead to false positives (i.e., 1.2% using MLP in our case).

3.5.2 Comparison of FCM with *n*-gram and BOFM

We evaluated the classification performance using n-gram (4-grams and 6-grams) and BOFM approaches for our dataset using 10-fold cross-validation. Since these methods use support vector machine (SVM) as a classifier in [33, 91], we also performed the evaluation using SVM. Table 3.4 compares the performance of our approach with n-gram and BOFM methods. The main limitation of n-gram techniques is that they have a large number of features. This will introduce substantial memory and performance overheads. In addition, they suffer from high false positives because they consider the strict ordering of system calls and therefore contain more common system calls about the file system operations that are also found in benign programs [91]. BOFM reduces the feature size, however, it does not consider the frequency of system calls and other rules (Rules 2 and 3, as explained

in Section 3.4), which can be exploited by malware to evade the protection. On the other hand, our FCM approach achieves considerable reduction in the feature size. Our FCM method significantly reduces the false positive rate (1.2%), while achieving marginally less true positive rate (97.6%), as compared to BOFM. Since the lower false positive rate is considered more important in malware detection, our approach has a better performance than the other two approaches. FCM has a lower false positive rate because we consider a more robust semantics of system calls, as explained in Section 3.4 (i.e., Rules 2-4).

3.5.3 Early Prediction Analysis Using FCM

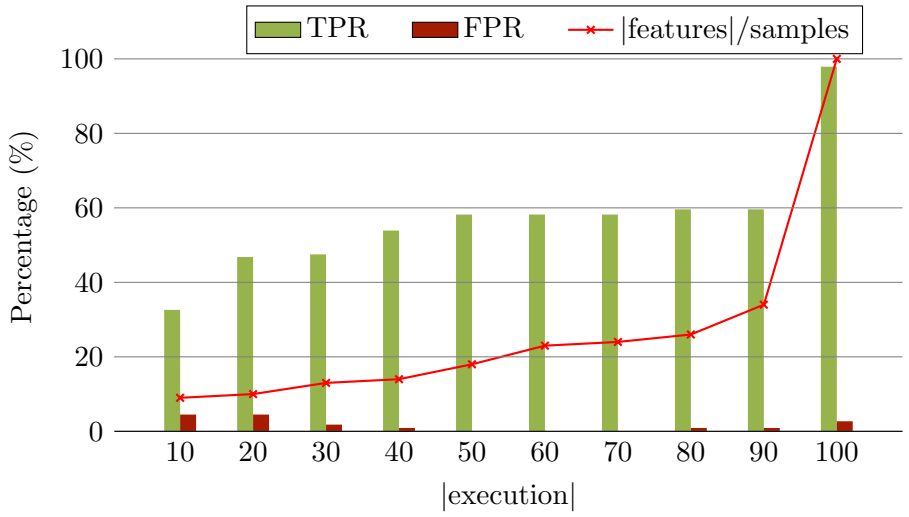
Early prediction is one of the key features of our approach, facilitating the detection of malware in real time. It is known that most of the malware performs malicious operations only at the end of their execution, while performing the debugging to retrieve system information, setting proper environment for malicious attacks in the early phase of execution. As shown in Fig. 3.1, first the system information is retrieved (lines 1-4), followed by setting up required conditions (lines 5-11) and finally the malicious activity is performed (lines 12-18). Since GuardOL performs the detection during the execution of the sample, based on the system calls traced from the running sample, it has the capability to detect malware during the early phase of execution (such as during the debugging or while setting up the environment). Thus, it may help to detect the malware before the stealthy operations have been performed.

Our approach uses MLP as the classifier. The MLP classifier is trained by a *Backpropagation* learning technique using the samples from set-A. As stated in Section 3.2, we test the early prediction using unknown samples, i.e., set-B. First we evaluate the performance of a classifier at different execution stages of the samples, i.e., at different percentage of execution (symbol $|execution|$ defines % of execution) of the samples. For this, we collect the complete trace of system calls for the samples during execution. Then we extract features from the initial system call traces by using a certain percentage of the complete trace. As shown

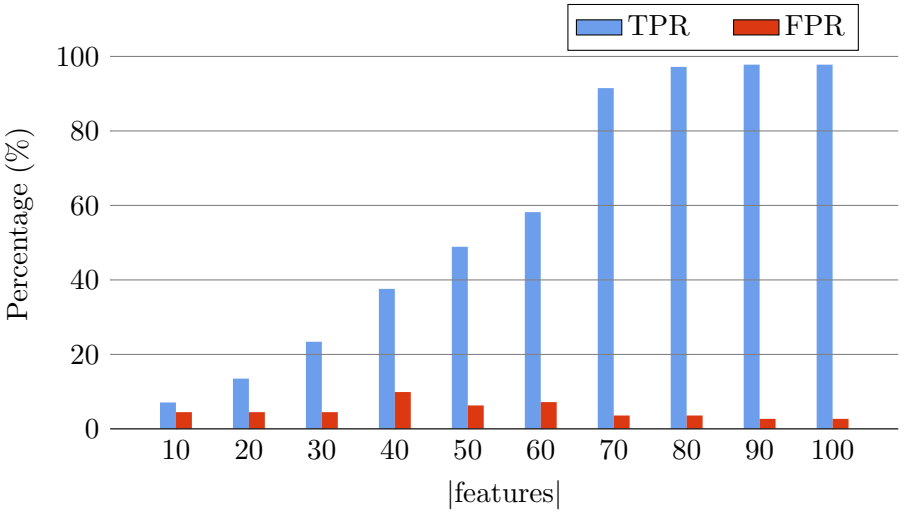
by the bar graph in Fig. 3.4(a), our method can detect more than 46% malware samples within the first 30% of their execution, with a false positive rate of about 2%. As the execution continues, more features are collected, which allows the detection of more malware samples. Finally, after the complete execution of the samples, more than 97% malware samples are detected, with less than 3% false positive. The early prediction rate is dependent on the features of the samples traced during the execution and their likelihood of being malware. As shown by Fig. 3.4(a), malware samples are mostly detected either at the beginning or after the complete execution of the samples. The line graph shows the percentage of features (symbol $|features|$ defines % of features) per sample at different execution stages. With the increase in $|features|$/sample, the TPR also increases. Between 90% and 100%, there is a significant increase in TPR. This is because the number of features increases at the end of execution of samples, which is shown by the line graph.

Second, we evaluate the performance of early prediction based on different percentage of features (symbol $|features|$ defines % of features) of the samples. For this, we collect the complete features from the samples. We test the early prediction using initial features at various percentage of the total features. As shown in Fig. 3.4(b), with the increasing percentage of features, TPR also increases. Our results show that using only 50% of the features of the samples, more than 48% of the malware samples can be detected (with false positive of 6%), while using only 70% of their features, more than 90% of the malware samples can be detected (with false positive of 3%). Overall, the results demonstrate the significance of the early prediction in malware detection.

As illustrated by Fig. 3.4, the early prediction can effectively detect malware only after the sufficient features are collected from a program, however, it cannot be based on the length of the executed trace. Since machine learning takes a few cycles to perform the classification, it is better to perform the classification operation only when the new features are extracted. This will help to reduce the overall computation and performance overhead. The behavior of malware varies across different malware families. Hence, there could not be a uniform policy for a



(a) With the increasing execution trace



(b) With the increasing features

FIGURE 3.4: Performance of Early prediction of malware

threshold number of features to be extracted before triggering the early prediction. Therefore, we suggest to trigger the early prediction once a new feature has been extracted.

3.6 Hardware Implementation

In this section, first we present the overview of our approach. Second, we present the architectural design of GuardOL and discuss its design considerations in detail.

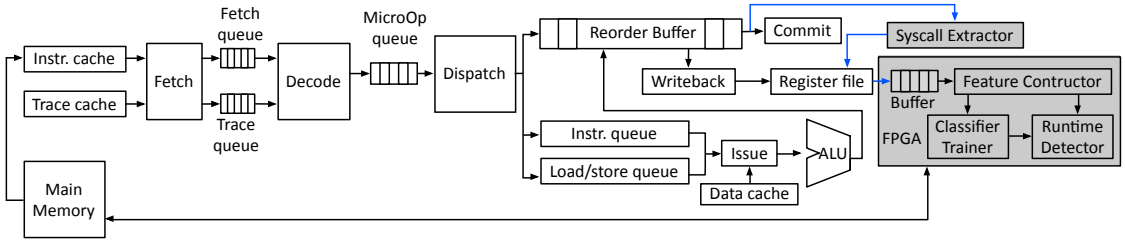


FIGURE 3.5: Architecture of GuardOL

3.6.1 Approach Overview

Fig. 3.5 presents the design of GuardOL. Our approach consists of two phases: *model building* and *runtime detection*. In the model building phase, we train our classifier engine using known malware and benign samples. In the runtime detection phase, the trained classifier is used to detect potential malicious behavior.

Model building phase contains the following three components:

Syscall Extractor logs system calls along with their arguments and return values invoked by the executed program, as shown by the example trace in Fig. 3.1.

Feature Constructor systematically extracts the meaningful features from system call traces, which are generated by Syscall Extractor, using our frequency-centralized model (FCM). And it embeds them into feature vectors as explained in Section 3.5.

Classifier Trainer uses the feature vectors constructed above to train the classifier engine, i.e., multilayer perceptron (MLP). In our architecture, the classifier training is performed offline and the trained data (including features and updated weights) is stored into the main memory (as shown in Fig. 3.5). When the system boots up, the trained data has to be fetched from the main memory and stored into the FPGA for runtime malware detection. We allow training of the classifier only when the dataset is updated with new malware samples.

Runtime detection phase leverages the Syscall Extractor to get the system call information from the Processor and uses Feature Constructor and **Runtime Detector** components to do the real-time prediction of the running application, where

prediction refers to classification of the unknown binary as either malware or benign. We will elaborate each component of GuardOL in Section 3.6.2.

In this work, we choose the platform consisting of Linux OS running on a 32-bit Intel x86 processor for generality. GuardOL utilizes OS-specific details for semantics of system calls and also to support multitasking operations. Our approach can also be ported to other OS and processor with the modification of the platform specification. Our architecture can support multitasking operations by tracing the control register CR3, as done in [27, 117]. Linux OS assigns a unique page directory for each process. Intel x86 architecture stores the physical address of base of page directory for the current running process in control register CR3. Upon context switching, the entry in CR3 is overwritten by the physical address of the new process. This CR3 register can be monitored for multitasking support. System calls can be appended with a unique identifier to differentiate the system calls obtained from different processes.

In this article, we prototype the GuardOL design in FPGA. As compared to ASIC, FPGA implementation offers lower cost, fast prototyping and the flexibility to adapt to the new malware samples.

3.6.2 Micro-architecture Design

In this work, we assume a System-on-Chip (SoC) architecture with a single core processor and an FPGA on one chip, as shown in Fig. 3.5. The interconnection between the FPGA fabric and commit stage of the processor is done at the design time of SoC. The FPGA logic can be updated in a trusted environment using a secured way, such as using bitstream encryption techniques [118]. Bitstreams can be stored along with the OS in main memory, as this region of the memory is secured and not accessible to the application programs. Our method assumes that the processor and the OS are trusted, but the programs running on them are not trustworthy.

As shown in Fig. 3.5, Syscall Extractor is integrated with the commit stage of the processor pipeline whereas, Feature Constructor, Classifier Trainer and Runtime

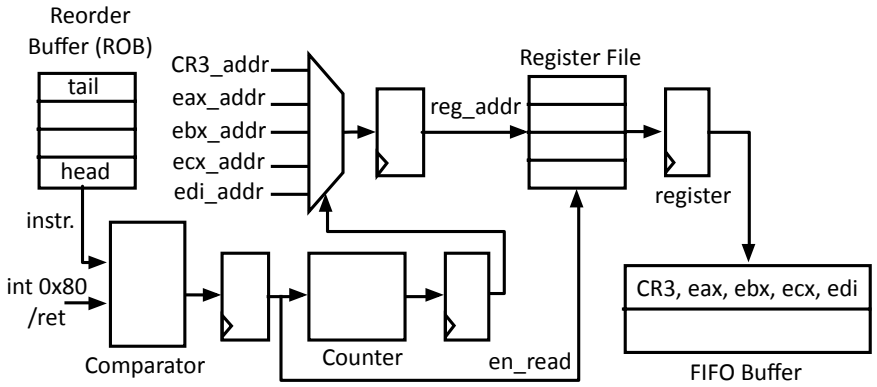


FIGURE 3.6: Syscall Extractor design

Detector are implemented on FPGA fabric. A special design consideration is to introduce a Buffer component. Because the processor operates at a higher speed than FPGA, and in addition, Feature Constructor also takes several cycles to perform its operations, adding a buffer can avoid the processor to stall and reduce the performance penalty.

Now we describe the implementation of each component in detail:

3.6.2.1 Syscall Extractor

Syscall Extractor traces the system calls augmented with their parameters and return values from the running program. In Intel ISA, system calls are invoked by using trap instruction `int 0x80` (or `sysenter`). As shown in Fig. 3.6, the system calls are traced from the reorder buffer (ROB) at the commit stage of the processor pipeline to identify the trap instruction. In x86 processor, for each of the instructions that enters the pipeline, ROB contains four fields: instructions, temporary register storage, result and validity of results. Once a trap instruction is committed in ROB, Syscall Extractor will read and store the system call type from register `eax` and other parameters from registers `ebx`, `ecx`, `edi`. For supporting multitasking operations, each entry in the buffer will be appended by the CR3 register value to identify the process. These register values will then be passed to the Buffer in FPGA. For some system calls, we also need to store their return values. Hence, once a following `ret` instruction is executed, its return value (given

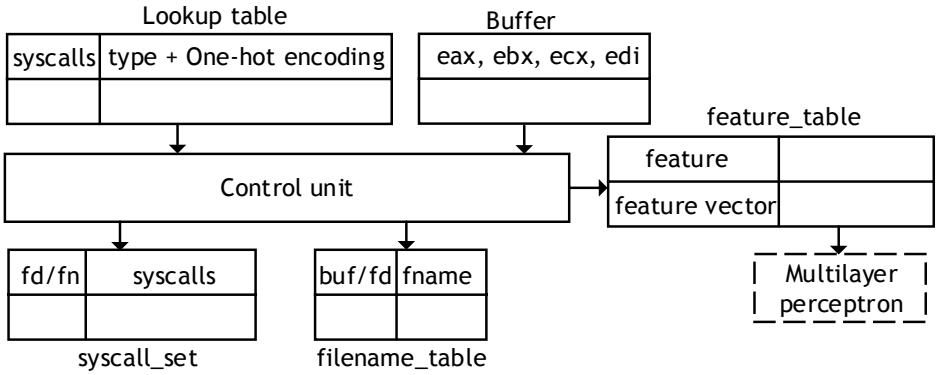


FIGURE 3.7: Feature Constructor design

by `eax` register) will also be stored into the Buffer. The system call information stored in the Buffer will be further processed by Feature Constructor. Note that since the extractor only needs to read specific registers from the register file, it will not interfere with the normal instruction execution. The extractor is pipelined to catch up with the processor speed for instruction checking.

3.6.2.2 Feature Constructor

Feature Constructor involves the identification of relevant system calls to construct system call sets (a.k.a. *features*), followed by feature vector construction. Feature Constructor consists of control unit, lookup table (LUT) and three data tables — syscall_set, filename_table and feature_table with each table implemented as key-value pairs, as shown in Fig. 3.7. LUT maps system calls into their types (represented by 4-bits) followed by one-hot encoded representation of system calls (64-bit). *Note that we only need to consider 64 security-critical system calls in our method to provide satisfying classification accuracy.* The syscall_set table stores relevant system calls with their corresponding filename/file descriptor, whereas filename_table stores filename with their corresponding file descriptor/buffer pointer. The system calls use the pointer to the filename, which is unique for each filename in a program. We use this pointer to represent the filename in our design. We implemented the first column of each of three data tables as Content Addressable Memory (CAM) to expedite the searching process whereas the second column as

conventional memory. The control unit is implemented as a state machine, which performs the fetching of system calls and parameters from the Buffer.

System call sets are constructed by grouping relevant system calls based on the rules introduced by FCM (in Section 3.4). To this end, we divide the security-critical system calls into 10 types, based on the semantics of system calls, their arguments and return value.

System calls with:

1. *first argument pointing to filename* (e.g., `chmod`),
2. *return value representing file descriptor* (e.g., `socket`),
3. *first argument as file descriptor* (e.g., `fstat`),
4. *no filename/file descriptor in arguments* (e.g., `uname`),
5. *resource-critical system calls* (e.g., `brk`),
6. `open`,
7. `read`,
8. `write`,
9. `close`, and
10. `mmap`

As stated above, the `eax` register is used to extract system calls (when the instruction is `int 0x80`) and return value (when the instruction is `ret`). Similarly, filename (fn) and file descriptor (fd) are given by `ebx` register; source/destination file (buf) for `read/write` system calls is given by `ecx` register; and file descriptor (fd) for `mmap` system call is given by `edi` register. System calls with common filename are grouped together and stored in syscall_set table (Rule 1). For memory mapping of a file or a device (Rule 2), `mmap` is separately stored corresponding to the file or the device in syscall_set table. filename_table maps the filename for the corresponding file descriptor. For `read` system call, filename_table stores the mapping of the content to the source file (Rule 3), which is used by `write` operation.

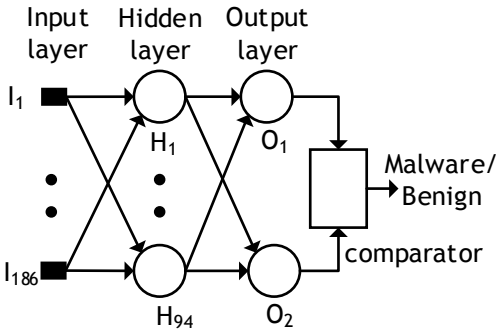


FIGURE 3.8: Runtime Detector design

feature_table stores features along with the corresponding feature vector, which consists of '1'/'0' or the frequency of a feature, as discussed in Section 3.5.

We emphasize that one-hot encoding representation of a system call is an important design consideration. As said in Section 3.4, we do not consider the order of system calls for feature construction. To store the set of system calls, either the sorting technique or the hashing technique is needed. However, they require a large amount of computation and resources. Using one-hot encoding technique, system call sets can be constructed using the *ORing* technique, yet meeting the requirement of unique representation and order independency.

3.6.2.3 Runtime Detector

It performs feedforward computation in a MLP network (as shown in Fig. 3.8), using the weights updated during model building phase, to classify a sample as a malware or a benign. The computation involved is given by Equation 3.1-3.2.

$$y_j^l = \sigma^l(s_j^l) = \sigma^l(\sum_{i=1}^{N_l} w_{ji}^l x_i + \theta_{ji}^l) \tag{3.1}$$

$$\sigma^l(x) = 1/(1 + e^{-x}) \tag{3.2}$$

where j is a neuron in layer l; s is the weighted sum; w_{ji}^l is the weight for the input i at neuron j; x_i is the input, θ_{ji}^l is the bias value; N_l is the total number of neurons in layer l.

At runtime, weights from the training are stored in the FPGA on-chip memory.

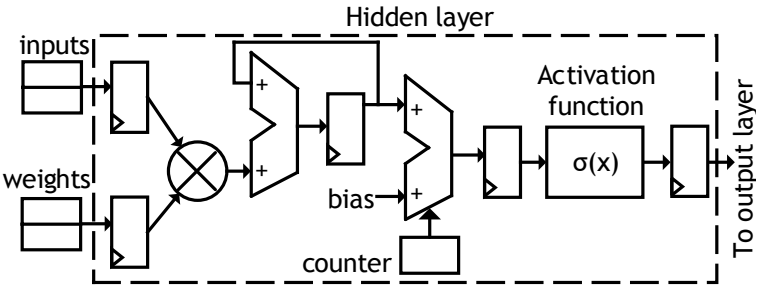


FIGURE 3.9: Implementation of feedforward logic

Feature vector generated by Feature Constructor is taken as the input to the MLP network. In our design, hidden and output layer neurons as well as multiply and addition (MAC) operations in each neuron are pipelined, as shown in Fig. 3.9. The output layer implements the same logic as the hidden layer.

Now we detail our design choices for MLP implementation:

Network structure: Based on our dataset, the MLP network has 186 inputs in the input layer, 94 neurons in the hidden layer and 2 neurons in the output layer, as shown in Fig. 3.8. Two neurons of the output layer determine the likelihood of the given sample being either malware or benign. We adopt the *winner takes all* strategy, i.e., if the output value at malware neuron (O_1) is greater than that of benign neuron (O_2), then the running sample is classified as a malware, otherwise as a benign program.

Activation function: Activation function is used for compressing the MAC output in each neuron between 0 and 1, given by $\sigma(x)$, as shown in Fig. 3.9. In general, *sigmoid* or *tanh* function is used as an activation function (Equation 3.2), which requires a large number of resources due to the exponential computation. We implement activation function using *piecewise linear approximation* (Equation 3.3) [116], for both hidden and output layers neurons, to save

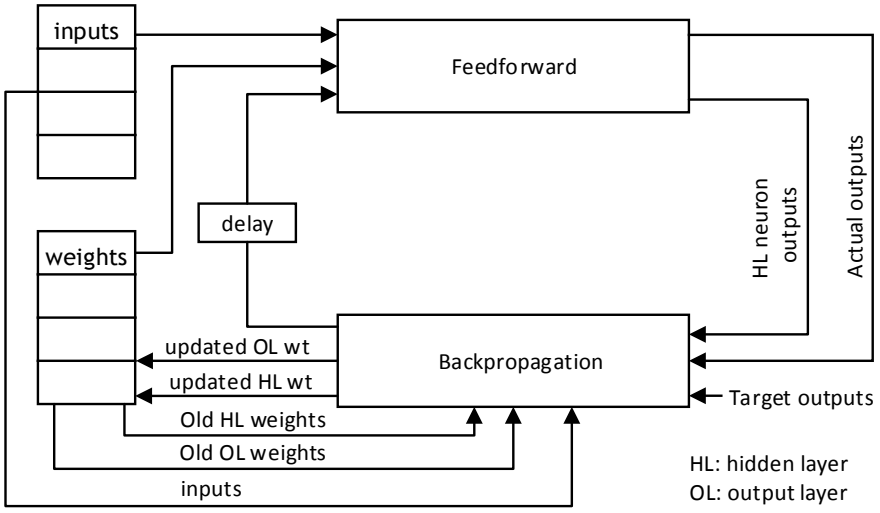


FIGURE 3.10: Classifier Trainer design

the hardware resources.

$$\sigma(x) = \begin{cases} 0 & \text{if } x \leq -8 \\ (8 - |x|)/64 & \text{if } -8 < x \leq -1.6 \\ (x/4 + 1/2) & \text{if } |x| < 1.6 \\ 1 - (8 - |x|)/64 & \text{if } 1.6 \leq x < 8 \\ 1 & \text{if } x \geq 8 \end{cases} \tag{3.3}$$

Number format, Data size, Precision: Arithmetic representation of inputs, weights and outputs of neurons have significant impact on hardware resource utilization [116]. Our design uses the fixed point implementation to save the hardware resources [116] while still maintaining the same classification accuracy. We implement our design using 1-4-11 format, with 1 bit for sign, 4 bits for integer and 11 bits for fraction. It has a high precision of 2^{-11}, which does not compromise the classification accuracy.

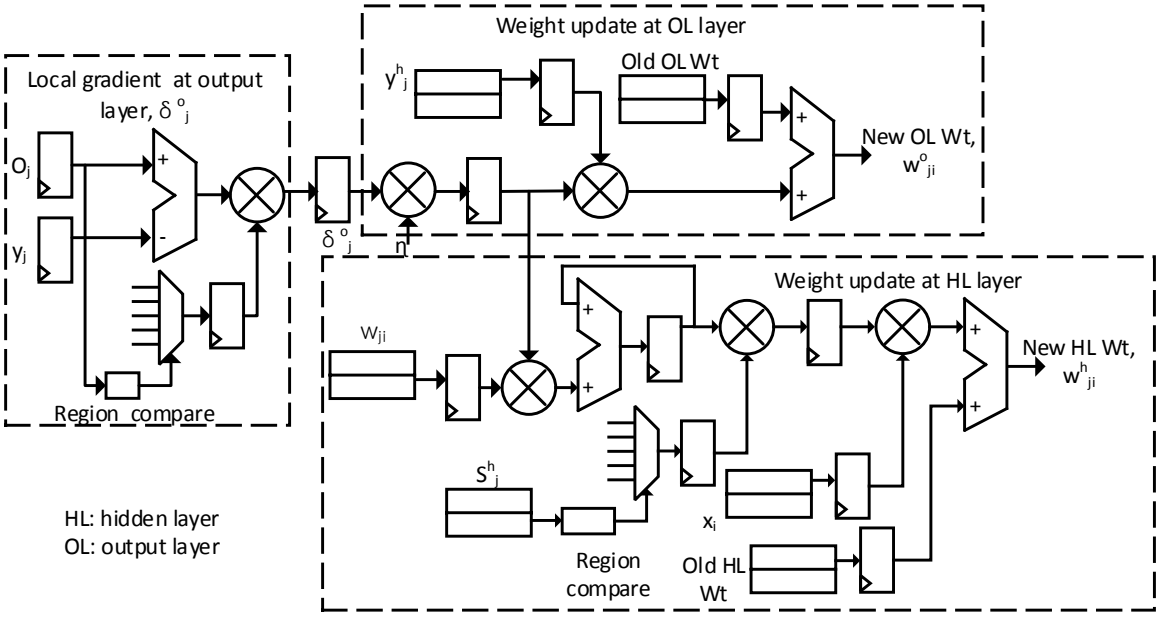


FIGURE 3.11: Implementation of backpropagation logic

3.6.2.4 Classifier Trainer

Model building phase involves the training of the MLP network using feature vectors, which are obtained from the training data set. We use *Backpropagation* learning technique for finding appropriate weights for different inputs at each neuron. Two types of computations — feedforward and backpropagation — are performed during training (as shown in Fig. 3.10), whereas only feedforward computation is performed during runtime using the trained weights. The detailed design of backpropagation computation is illustrated in Fig. 3.11, whereas feedforward computation in Fig. 3.9.

All the weights are updated as:

$$w_{ji}^{l}(n+1) = w_{ji}^{l}(n) + \eta \delta_{j}^{l} y_{i}^{l-1} \tag{3.4}$$

$$\delta_{j}^{l+1} = \varepsilon_{j}^{l+1} \sigma_{1}^{l}(s_{j}^{l+1}) \quad \text{for} \quad l = 1, 2, ..., L \tag{3.5}$$

$$\varepsilon_{j}^{l} = \begin{cases} O_j - y_j^l & \text{for} \quad l = L \\ \sum_{i=1}^{N_{l+1}} w_{ji}^{l+1} \delta_j^{l+1} & \text{for} \quad l = 1, 2, ..., L-1 \end{cases} \tag{3.6}$$

where, $\delta_j{}^{l+1}$ is the local gradient calculated based on the error ε, η is the learning rate, s and y_i^{l-1} are obtained from the feedforward computation in Equation 3.1, σ_1^l is the first derivative of $\sigma(x)$ given by Equation 3.3 in layer l, O is the target output at the output layer neuron and L is the final output layer.

During training, error is calculated at the output layer and propagated backwards towards the hidden and input layers, as shown in Fig. 3.10. The design is fully pipelined to improve its throughput. The computation of local gradients at output layer neurons (Equation 3.5) takes few cycles and thereafter, the weights at the output and hidden layers are updated in parallel (as depicted by Fig. 3.10). Feedforward computation runs in parallel to the backpropagation, once the hidden layer weights begin updating. Backpropagation reads the old weights and updates the new weights into FPGA on-chip memory.

3.7 Evaluation

3.7.1 Experimental Setup

To evaluate our design, we implemented Feature Constructor, Classifier Trainer and Runtime Detector on FPGA device of Virtex-5 LX110T using Xilinx ISE 14.5. Power consumption for FPGA logic was estimated using Xilinx Xpower Analyzer tool. Note that we use Virtex-5 device only to demonstrate GuardOL architecture. The real design of GuardOL has very small footprint with only required reconfigurable fabric. Syscall Extractor requires changes in the processor core (see Fig. 3.6). We used Synopsis Design Compiler tool to evaluate the area and power for Syscall Extractior at 65 nm technology.

To perform the function verification and performance estimation of the whole design, we integrated the models for the hardware components (frequency and number of cycles) into a cycle-accurate processor simulator, Multi2sim, which was configured as 32-bit, x86 @2.66GHz to mimic the real processor. Then the

Table 3.5: Memory requirement

	#entries	width (bits)	size (bytes)
Buffer	10	160	200
Lookup table	64	100	800
syscall_set	300	96	3600
filename_table	225	64	1800
feature_table	186	80	1860
Weight	17768	16	35536
Total			43796

instruction trace with extracted system calls was given as the input to the modified simulation platform.

3.7.2 Experimental Results

3.7.2.1 Syscall Extractor Cost Evaluation

The results show that Syscall Extractor has a negligible area overhead of 0.003% (3588.839 μm^2) for the processor, while consuming a very small power of 2.0762 mW. This reflects that the modification in the processor is minor. Moreover, since the logic connected to the register file is simple as shown in Fig. 3.6, i.e., two registers to drive a 4-to-1 multiplexer (MUX) and a small first in first out buffer (FIFO), the extra delay added to the processor pipeline is also small (less than 2%).

3.7.2.2 FPGA Implementation

Table 3.5 summarizes the overall memory requirement including Feature Constructor and MLP design (Classifier Trainer and Runtime Detector), which is less than 43 KB. Weights require more memory compared to other components (>34 KB)

as the number of weights required by our design is 17768, each of 16-bits. The estimation has been given for one process. While monitoring multiple processes concurrently, the memory usage will be increased for Buffer, syscall_set, filename_table and feature_table. As per our results, the usage for these components is 7 KB per process, while weight can be shared by all the processes. Assuming the memory on modern FPGA to be in MB, it will be sufficient for multiple processes (e.g., 1 MB will be sufficient for 140 concurrent processes, including weight). Table 3.6 presents the results for resource utilization in hardware and latency for Feature Constructor and MLP in FPGA. It can be seen that the resource requirement of the logic implementation is also low.

As mentioned before, we placed a buffer between Syscall Extractor and Feature Constructor to reduce the performance penalty due to the slow speed of FPGA. Since we trace only system calls, the size of the buffer is dependent on the gap between appearance of two successive system calls in the program and also on the processing speed of Feature Constructor. As per our results, the gap between system calls in the instruction stream is large, such that even a small buffer size (set to 10 in our design) can hold the system call information while FPGA continues its processing.

About the performance, our Feature Constructor takes 3-12 cycles (an average of 10 cycles) to complete its operations, depending on the type of system calls as discussed in Section 3.6.2.2. The MLP takes 17502 cycles on FPGA for one sample, both for training (running at 232 MHz) and detection (running at 250 MHz). As shown in Table 3.6, the dynamic power consumption for FPGA design, including Feature Constructor, Classifier Trainer and Runtime Detector is very low. Note that as more digital signal processing blocks (DSPs) are used in our design, the performance can be significantly increased due to the parallel processing of the samples with the cost of increase in area and power. The DSP blocks occupy around 10% of the total logic area (i.e., area of 2 DSPs$\approx$5 CLBs[1]) and consume 1% of the total dynamic power on FPGA. Hence, when doubling the number of

[1]CLB: Configurable logic block

Table 3.6: Performance of FPGA logic units

	Metrics	Feature Constructor	Classifier Trainer	Runtime Detector
Resources	Flip flops	625	537	253
	LUTs[2]	1254	770	396
	DSPs	-	8	2
Performance	Cycles	10 (avg.)	17502	17502
	Frequency (MHz)	200.56	232	250
	Execution time (μs)	0.0498	75.44	70
Power	Dynamic Power (W)	0.17841	0.18170	0.08638

DSPs, the logic area and power increase around 10% and 1% respectively, but the throughput of MLP can be almost doubled.

Based on our understanding of system calls, we selected small buffer size (of 10 entries) in our experiments. We found no overflow for this buffer size because each system call execution takes large number of cycles, while the feature construction takes only about 10 cycles for its operation. For a processor running at 2.66 GHz and FPGA at 200 MHz, 10 FPGA cycles correspond to 130 processor cycles. The syscall invocation (using trap instruction) is followed by syscall handling subroutine and appropriate syscall handler, which take more than 130 cycles for execution. Thus, the buffer size of only one entry is required. But, to leave a safe margin, we selected the buffer size of 10 entries, which has no overflow in our experiments. Therefore, a small buffer size can hold the information while feature construction operation is performed.

3.8 Summary

We presented GuardOL, a hardware-enhanced architecture to detect malware at runtime. Our approach first extracts the system calls and constructs the features

[2]LUT: Lookup Table

based on the high-level semantics of malicious behavior. To this end, we propose a novel frequency-centralized model for feature construction. The features obtained from the benign and malware samples are then used for training the machine learning classifier, multilayer perceptron, which is used to detect the malware samples at runtime. The evaluation results show that GuardOL is fast, effective and has a marginal performance overhead on the processor. In future, we will extend our approach to multicore systems.

4

Online Malware Defense using Attack Behavior Model

4.1 Introduction

With the arms race between adversaries and security personnel, malware techniques have continually evolved. Recent malware uses sophisticated techniques to evade the detection technique and to hide its malicious intents. Meanwhile, signature-based detections (e.g., antivirus, scanners) are incapable to defend against the advanced techniques. Hence, effective anti-malware solution must be developed and improved constantly to protect the computing systems.

A substantial amount of work has been done for malware defense. The widely used commercial solutions (antivirus, scanners) use static techniques (e.g., using signature [12]), which are efficient, but can be easily evaded by malware variants and advanced techniques (e.g., code encryption). While dynamic approaches, based on function call monitoring [28], dynamic binary instrumentation [89], virtual machine introspection [88], are proposed to improve the accuracy of the detection. For example, [91, 92] use features obtained from malware samples to train the machine learning algorithm in order to model the malicious behavior. However, these techniques [28, 88, 89, 91, 92, 119] all suffer from the high performance overhead during detection due to large number of computations involved.

Deterministic Finite Automaton (DFA) is proposed as an effective model for capturing the malicious behavior [38]. In comparison to other dynamic approaches [28, 88, 89], DFA-based technique has low performance and resource overhead and has the ability to detect malware and its variants. However, existing DFA approaches are domain specific (e.g., JavaScript malware [38]). Moreover, they are implemented in software only, which can be bypassed or disabled by advanced malware. Malware often uses sophisticated evasion techniques (e.g., anti-debugger, anti-ptrace) to bypass software-based malware detection solutions. Moreover, kernel-level rootkits and bootkits can gain OS level (Ring 0) privilege to disable the software detection. Recently, hardware techniques have been recognized as a promising approach to defend against malware.

In this chapter, we propose a software-hardware combined approach to build an online malware defense at low performance and resource overhead. Our approach consists of two phases: *offline learning phase* and *runtime detection phase*. We observe that malicious behaviors are reflected by a sequence of security related system calls. During the offline learning phase, we use a software approach to learn the DFA models of malware such that each DFA summarizes the common system call execution patterns of one type of malware. We use L* algorithm [120, 121] to learn the DFAs from the system call patterns. As a novel contribution, we also incorporate frequency as a parameter to detect those malicious behaviors that execute repetitive patterns to do anomaly in the system. During runtime detection

phase, our hardware-enhanced architecture utilizes the learnt DFA models to detect malware by checking whether the program execution contains the system call sequences in any DFA, i.e., if the system call sequence in the execution trace can be accepted by a DFA. Our hardware-enhanced architecture is named as *malguard* (malware guard). We develop an architectural design of malguard integrated with the processor pipeline which can do parallel detection of multiple DFAs.

We evaluate our method using real world data of 168 Linux malware samples and 370 benign applications. Our evaluation shows that: 1) malguard can detect malware and its variants with high accuracy and no false positive; 2) it is able to classify malware samples based on their attack behavior and detect zero-day malware with similar attack patterns; 3) it detects malicious behavior in real time (within 5 clock cycles), which implies that it can possibly prevent the system from major damages caused by the malware; and 4) it has a small footprint.

The rest of the chapter is organized as follows: In Section 4.2, we discuss the related work. Section 4.3 describes our approach. We discuss the hardware design in Section 4.4. We present our evaluation results in Section 4.5 and finally conclude in Section 4.6.

4.2 Related Work

Recently, system call based approaches have been widely adopted for modeling malicious behavior [28]. Many of them employ machine learning techniques to model the malicious behavior [91, 92]. These techniques have better performance in malware detection, as compared to other dynamic approaches. However, they still incur high performance and resources overhead during runtime implementation due to the large number of computations/resources involved. These limitations hinder their implementation for runtime defense. Demme et al. [101] used hardware performance counter to detect malware but they have a high false positive rate. Das et al. [122] proposed a machine learning based hardware-enhanced architecture to reduce the performance overhead. But their detection is slower (i.e.,

>1000 clock cycles) because of the heavy computations involved in the machine learning technique.

Xue et al. [38] proposed a DFA-based approach for modeling attack behavior on the browser. Their work is specific to Javascript malware only, and they focused on the offline analysis rather than online defense. Rahmatian et al. [117] proposed a host-based intrusion detection using FPGA, which is closely related to our work. This approach used the system call sequences to build a finite state machine (FSM) to model the behavior of benign programs. However, it has several limitations. First, each program needs to be profiled to generate corresponding FSM. It does not allow unknown programs to run on the system. Second, the FSM can be significantly large for real-world applications. This will require a large memory. In contrast, our approach uses DFA for attack patterns only, which requires only a small amount of memory. In addition, our method does not require profiling beforehand and therefore, it allows unknown programs to run on the system.

4.3 Malware Modeling and Learning

In the offline learning phase, we first run the malware samples in a sandbox to extract the traces of system calls invoked by the malware. Then we use L* algorithm to learn the DFA of the traces. The L* algorithm is implemented in the software because it is more flexible than hardware.

4.3.1 Attack Behavior Modeling

We build DFAs for the following seven popular attack behaviors that are observed in our malware dataset:

I. Virus replication behavior – Virus replicates itself by first copying its binary into the temporary/executable file and finally executing the file.

II. Overloading system memory – Some malware repetitively executes system calls to increase the program memory size to overload the main memory. This often results in crashing of OS and degrading the performance.

III. Scanning ports and executing shell commands – This malicious behavior includes scanning ports and receiving malicious payloads from internet (e.g., to execute shell commands).

IV. Data leakage on the network – This attack captures the malicious behavior that leaks the information (e.g., browser cookies) from the host onto the network.

V. Spawning – A typical behavior of virus is to replicate itself by creating several clones (child processes) from its own binary.

VI. CPU resource overloading – This attack intends to control the CPU usage by performing anomalous activities, e.g., modifying system files, creating harmful files. This makes the system extremely slow leading to CPU-starvation attacks.

VII. Killing other running processes – It subsumes the malicious behavior that intends to kill other running processes. Typically, malware first changes the default actions for the given signal (e.g., interrupt) and then kills running processes, which is not noticed by OS.

Although DFA can accurately model the malicious behavior, DFA-based detection usually incurs false positives due to the fact that some malware executes similar system call patterns as the benign programs while their maliciousness is resulted from their repetitive pattern. For example, `brk` system call is executed by benign program to increase the program data segment size, while, "Brk.c" repetitively uses `brk` to overload the system memory, which may lead to CPU starvation attacks. Another malware "Spyeye collector" sends data from host system to the network repetitively. Hence, for such cases, DFA model alone can result in high false positives (shown in Section 4.3.3). In order to consider such malicious behaviors (attack type II and VI), we combine DFA with frequency to model

the repetitive pattern. The threshold frequency at which a behavior should be considered malicious is obtained by heuristics.

4.3.2 Offline Behavior Learning

We use the L* algorithm [120, 121] to learn the DFA models for describing the attack behaviors of malware. Due to space limit, we only outline a high level description of L* algorithm. The L* algorithm is an active DFA learning algorithm which assumes the presence of a *teacher* and the finite set of alphabet symbols Σ for the DFA. The teacher is assumed to know the DFA U, which is unknown to L* in the beginning. The teacher is able to answer *membership queries* and *candidate queries* asked by L*. A membership query asks the teacher whether a given string tr is accepted by the DFA U. The teacher must answer $true$ or $false$ honestly according to the DFA he/she knows; a candidate query asks whether a candidate DFA C generated by L* is equivalent to U. The teacher answers $true$ if $C = U$; however, if the two DFAs are not equivalent, besides answering $false$, the teacher also provides a counterexample string ce which identifies the differences between the two DFAs, i.e., $ce \in (C - U) \cup (U - C)$. Based on above assumptions, [120] proved that L* learns the unknown DFA U in polynomial time.

For malware detection in our context, we use the set of system calls as the alphabet symbols Σ in L*. To answer a membership query, the teacher checks whether a given trace of system calls is malicious or not; to answer candidate query with a given candidate DFA C, the teacher checks whether C describes the malicious behavior. The only inputs we have are a set of malicious traces and a set of benign traces collected by running the malware samples in a sandbox. However, it is highly possible that the strings (i.e., traces of system calls) asked by L* for membership queries are not in the input traces. Thus, the teacher cannot answer all membership queries by using the input traces only. There are two issues related to answering candidate queries: 1) the teacher only knows a set of traces for malware and benign programs and it also does not know the DFA model

(i.e., U) for attack behavior; 2) the teacher needs to provide a counterexample if the candidate DFA is not good enough.

We use pairs of precedence rules in the form (a, b), where $a, b \in \Sigma$ and $a \neq b$, to solve above issues in answering membership queries and candidate queries. The set of rules are generated automatically from input traces. For each attack type, we use its malicious traces as the $MSet$ and a set of benign traces as the $BSet$. For each malicious trace, we collect all pairs of consecutive system calls into $MPSet$ and collect all pairs of consecutive system calls from each benign trace into $BPSet$. The elements that are in $MPSet$ but not in $BPSet$ (i.e., the set difference between $MPSet$ and $BPSet$) are considered as the precedence rules for a particular attack.

4.3.2.1 Answer Membership Queries

We use precedence rules to answer membership queries. For a membership query with trace tr, we use the set of precedence rules as a filter to check whether tr is malicious or not and answer the membership query accordingly. If the trace violates any of the precedence rules for this attack type, we return $false$, i.e., the trace is not malicious. If tr passes all the precedence rules, then tr is considered as malicious and the answer is $true$. Note that a trace tr passes a precedence rule (a, b) only if tr contains both a and b, and b comes after a in tr.

4.3.2.2 Answer Candidate Queries

For answering candidate queries, we use the benign and malicious traces to check whether there is any inconsistency between the traces and the candidate DFA C. For a benign trace tr, we run the trace on the candidate DFA C and check whether the final state of running tr on C is an accepting state or not, i.e., $tr \in C$?. If $tr \in C$, then there is no inconsistency; however, if $tr \notin C$, then there is an inconsistency. This is because according to our available input traces, tr is benign, but according to the candidate, tr is malicious. In this case, we return $false$ with tr as the counterexample. For a trace tr' in the malicious input traces, we run it

on DFA C and check if $tr' \in C$. If $tr' \in C$, no inconsistency is found; however, if $tr' \notin C$, then we return $false$ with tr' as the counterexample trace.

Since the size of the input traces for each attack type is small, it is often inadequate to check the candidate DFA against the input traces only. We also generate test traces up to a specific length and check them against the set of rules and the candidate DFA to find any inconsistency. The number of possible traces are exponential. To alleviate this problem, we calculate the transitive closure of the set of precedence rules. During the generation of test traces, we discard those traces which violate any precedence rule in the transitive closure *explicitly*. A trace tr violates a precedence rule (a, b) explicitly if tr contains both a and b but b occurs before a in tr. We check each test trace against the filters and also run the trace on the candidate DFA. If the results of running the trace on DFA and the filters are different, we return the trace as the counterexample. If no inconsistency is found in the above two steps, we return $true$ to the candidate query. Then the final candidate DFA is equivalent to the candidate DFA C.

4.3.3 Evaluation of Offline Learning

To show the effectiveness of the proposed approach, we evaluated our dataset with 7 DFA models (corresponding to the 7 attack types). Our dataset consists of 168 malware collected from [96] and 370 benign samples. For each attack type, we used 50% of the samples for learning the DFA and remaining 50% for testing. As shown in Table 4.1, our approach can detect all malware test samples (numbers inside the parentheses represent total test samples). To evaluate the false positives, we tested benign samples by using DFA without frequency and with frequency. Our results show that, for attack type II and VI, our approach has combined false positives of >6%. Using frequency along with DFA, it reduces to zero.

Table 4.1: Detection Performance of Malguard

Attack	Malware		Benign	
Behavior	Train-set	Test-set	W/o freq.	Freq.
Type I	14	14 (14)	0	0
Type II	22	21 (21)	9	0
Type III	5	5 (5)	0	0
Type IV	6	5 (5)	0	0
Type V	6	6 (6)	0	0
Type VI	30	29 (29)	15	0
Type VII	3	2 (2)	0	0

4.4 Hardware Implementation

In this section, we discuss the architectural design of malguard followed by its evaluation.

4.4.1 Microarchitecture Design

As shown in Fig. 4.1, malguard runs in parallel to the processor. For the proof of concept, we implement the malguard design in FPGA. It can also be implemented in ASIC platform. FPGA supports flexible update of the hardware verification logic by utilizing its reconfigurability feature. The malguard is integrated with the commit stage of the processor. We assume that the processor and OS are trustworthy but the user applications may be malicious. Our platform consists of 32-bit x86 processor running Linux. Henceforth, some details are platform specific, however, our design can also be implemented on other platform with change in specification.

Our design consists of three main components:

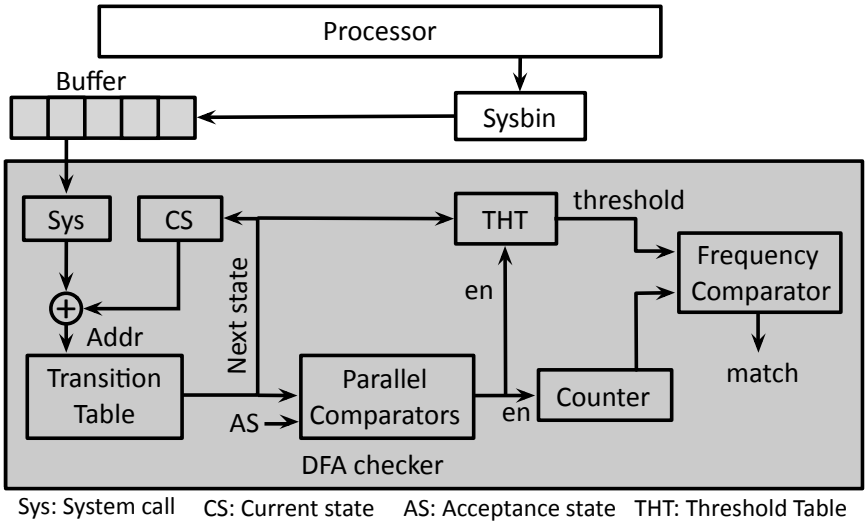


FIGURE 4.1: Malguard Design

4.4.1.1 System call bin (sysbin)

Sysbin is integrated with the commit stage of the processor pipeline. In the commit stage, sysbin traces the software trap instruction (`int 0x80` in x86) from the reorder buffer. At trap instruction, the sysbin reads the system call type from the `eax` register, as done in [122]. The system call is pushed into the buffer (Fig. 4.1), which is later used by the DFA checker. In order to differentiate the system call traces obtained from various processes (for multitasking support), we trace CR3 control register. Sysbin requires minor modification in the processor design, as described in [122].

4.4.1.2 Buffer

Buffer is an important design consideration to reduce the overhead on the processor. As the DFA checker takes a few cycles to complete its operation, Buffer is used to avoid the processor stalls and reduce the performance penalty.

4.4.1.3 DFA checker

It obtains system calls from the Buffer and detects malicious behavior by validating the transition of states. As shown in Fig. 4.1, it consists of a transition

table and a control logic to detect the malicious behavior. The transition table stores all the state transitions, which are obtained from the *offline learning*. As discussed in Section 4.3, we use a frequency parameter to differentiate between benign and malicious traces for a few attack patterns (Type II and VI). Based on the system call (sys) and current state (CS), the next state (NS) is obtained from the transition table. Parallel comparators compare NS with the accepting states of the DFAs (stored in the FPGA registers). If NS is an accepting state, the counter is incremented once and the corresponding threshold frequency is obtained from threshold table (THT). Then frequency comparator compares the threshold frequency and the counter value. For those attack behaviors which do not have the repetitive pattern, we store its threshold frequency as 1. Once the threshold frequency matches to the counter value, we flag the program as malicious. Our design can handle multiple DFAs. The CS of each DFA is initialized to its initial state. During checking, we check whether sys matches with each DFA on its CS. If there is a transition (CS, sys, NS) in the DFA, we change its CS to NS; if there is no such transition, the CS of the DFA is not changed.

4.5 Evaluation

4.5.1 Experimental Setup

We implemented DFA checker on FPGA device of Virtex-5 (LX110T) using Xilinx ISE 14.5. To evaluate the area of sysbin, we used Synopsis Design Compiler tool at 65 nm technology. To verify the functionality and to estimate the performance of the overall design, we integrated the hardware components (frequency and cycles) in Multi2sim, a cycle-accurate processor simulator, and configured it to 32-bit, x86@2GHz. The system calls were traced during program execution on the simulator.

4.5.2 Experimental Results

For the proof of concept, we implemented sysbin and DFA checker. Our results show that sysbin has a negligible area overhead of 0.003% (3588.84 μm^2) for the processor. DFA checker was implemented in FPGA. Since Linux has above 300 system calls, we used 9-bits for representing the system calls. In total, there are 427 transitions for 7 attack types, so we use 9-bits to represent current state register. The transition table has a depth of 427 corresponding to the number of transitions, while each row is of 9-bits (to represent the states). As shown in the Table 4.2, the memory requirement is just 481 bytes. The memory requirement is very low as compared to [117]. This is because we only store the DFAs for attack behavior, while [117] stores DFA for a whole benign program. Since the system call traces can be very large for real world applications, the memory requirement will be substantial for their method. Moreover, their approach requires DFA for each program, which will require a large amount of memory.

About the performance, our DFA checker takes 4 clock cycles running at 265 MHz to detect the attack behavior, while their method [117] takes 3 clock cycles to detect anomaly. It is a marginal difference, which is not significantly important. The malguard has low resources utilization on the FPGA (<1% on Virtex-5). We selected a small buffer size (10 entries) in our design to hold the system calls while the checker performs the validation. The smaller buffer size is sufficient because the gap between system calls is large enough so that the checker can complete its operation before the buffer gets filled up.

4.6 Summary

In summary, we proposed a DFA-based online malware detection approach. We presented the design of malguard, which detects malware at runtime. We also introduced a frequency parameter to model repetitive malicious behavior. The evaluation shows that our approach has a high detection rate with no false positive.

Table 4.2: Resource Utilization of DFA Checker

Resources	Flip flops	21 (<1%)
	LUTs	18 (<1%)
	BRAMs	1 (<1%)
Performance	Cycles	4
	Frequency (MHz)	265
Memory		481 Bytes

The malguard facilitates real time malware detection with low performance and memory overhead. In future, we will extend our approach to learn the attack pattern with repetitive system calls automatically.

5

Control Flow Integrity Approach Against Runtime Memory Attacks

5.1 Introduction

Embedded systems have proliferated at present and have been prevalent in a wide range of applications, such as consumer electronics, wearable devices, medical instruments, gaming consoles, industrial control systems, military devices. Security is a growing concern in these systems due to the presence of amplitude of private and critical information. Recently, adversaries have primarily targeted at these devices to get the private data of the users or to perform malicious actions [123, 124].

The primary challenge in embedded systems security is the resource constraints – limited computation, battery and storage. As a result, desktop security solutions cannot be implemented on the embedded platform, as they have significant performance overhead and also require substantial amount of resources. Therefore, embedded systems security demands lightweight and yet effective solutions. One of the main reasons that these systems are susceptible to exploits is due to the use of unsafe languages such as C/C++. Runtime attacks such as buffer overflow based stack smashing and code reuse attacks (CRAs) are common due to lack of bounds checking in these languages. Though Data Execution Protection (DEP) [111] can protect against the stack smashing, which requires code injection, CRAs such as Return Oriented Programming (ROP) [65] and Jump Oriented Programming (JOP) [68] can still be performed and are predominant currently. In general, ROP and JOP use small sequences of instructions (called *gadgets*) present in the existing code to perform arbitrary computation, thus avoiding the need for code injection.

Control flow integrity (CFI) enforcement is a promising approach to protect systems from runtime attacks. During the execution, CFI enforces the program to follow control flow graph (CFG), which is predetermined at the compilation time. Any deviation of the program execution from the normal CFG is considered as an attack. Unlike DEP, a proper CFI enforcement can defend against stack smashing attack as well as ROP and JOP, because these attacks deviate the control flow of the program from the normal CFG.

Although several CFI techniques have been proposed since its introduction by Saxena et al. [125], most of them either suffer from high performance overhead or have their own limitations. Binary instrumentation techniques [72] increase the code size and thus have high performance overhead. Some CFI implementations require the modification of ISA [77] or the presence of relocation information in binary [73]. Zhang et al. [126] proposed a CFI technique for complex libraries, which overcomes the previous limitations and also reduces the performance overhead; however, it only provides coarse-grained CFI, which is prone to advanced exploits.

In this chapter, first we present a novel approach to enforce fine-grained CFI at basic block level, named BB-CFI, which is targeted to defend buffer overflow based runtime attacks on memory, in particular stack smashing and code reuse attacks (e.g., ROP and JOP). Basic block (BB) [127] is defined as a sequence of instructions, having a single entry and a single exit point. In summary, BB-CFI policy allows:

- Function calls to target at the function entry, i.e., the first BB of the function
- Function returns to target at the instruction following the function call
- Indirect jumps to target at the starting address of a BB

There are some exceptions to the three policies, e.g., in multithreading, *longjmp()*, C++ exception handling and in the cases where returns are used as jumps. In this work, we handle these exceptional cases, by proposing extended rules to our BB-CFI policy.

Our approach comprises two steps: 1) the first step performs an offline *profiling* of the program in order to extract the control flow related basic block information; 2) the second step involves the runtime control flow checking to enforce BB-CFI policy using the extracted basic block information. Second, we present the design of a control flow checker (CFC), which tracks the program execution during the runtime to enforce BB-CFI policy. Thus, the CFC can detect any attempt to deviate the normal program execution. We implement the CFC design in FPGA as the proof of concept, which can provide high verification speed, low development cost and reconfigurability for post-fabrication functionality upgrade when needed. Nonetheless, the same design of CFC can also be implemented in ASIC platform.

Our method can detect all the control flow attacks from RIPE benchmark [128] that successfully run on our simulation environment. For evaluating the performance of our approach, we conducted the experiments on different scale of benchmarks, including SPEC CPU 2006. Our results show that BB-CFI eliminates >99% of the gadgets that are found by widely used ROPGadget tool [129]. The

BB-CFI policy has an average indirect target reduction (AIR) [126] of >99% and 100% verification accuracy. The CFC implementation on FPGA shows <1% performance overhead and low power consumption of approximately 78 mW, with a marginal area footprint. The overall results prove the effectiveness of the proposed BB-CFI method.

In a nutshell, our contributions in this work are:

- Proposal of a novel and fine-grained BB-CFI policy to defend stack smashing and code reuse attacks,
- Proposal of extended rules to handle exceptional execution flows, and
- Design and implementation of CFC on FPGA.

The rest of the chapter is organized as: In Section 5.2, we discuss the runtime attacks on memory. Section 5.3 reviews the related works. We present our approach in Section 5.4. Section 5.5 describes the proposed architecture and analyzes the security limitations. We discuss experimental results in Section 5.6 and finally conclude in Section 5.7.

5.2 Runtime Memory Attacks

Adversaries exploit vulnerabilities present in the software to perform runtime attacks on memory, such as buffer overflow based stack smashing and code reuse attacks. Buffer overflow based attacks are prevalent in embedded systems because the software running on them mainly use native languages – C/C++ and assembly – which lack bounds checking for the variables. In addition to this, the increasing complexity and connectivity in these systems result in increased vulnerabilities and a large attack surface, consequently providing ample opportunities for performing attacks. In general, the attackers exploit the vulnerabilities in the victim software by gaining the access to the dynamic data segment such as the stack or heap to overflow the allocated data space of the variables. In the stack smashing, first the

attackers inject their own code by providing malicious inputs to the variables and then overwrite the return address in the stack to redirect the control flow to their injected code. While in the case of code reuse attacks (CRAs) – ROP and JOP – the attackers use small instruction sequences, called *gadgets*, present in linked library or the victim program. The adversaries first create the payloads, consisting of gadgets or the code pointers that target the gadgets in the library or victim program. Then, they execute these gadgets in an orchestrated order to perform malicious operations. Though, DEP or W⊕X protection [111] (which implements non-executable memory) can thwart the code injection attack, CRAs can still be accomplished, as they use existing code sequences instead of the injected code.

In this work, we assume that an adversary can gain access to the stack or heap section of memory to perform runtime attacks – stack smashing, ROP and JOP. For the stack smashing, we assume that the DEP protection is either not implemented in the system or bypassed by the advanced technique [130], and therefore the attacker can inject the code. We also assume that the attacker is able to bypass the Address Space Layout Randomization (ASLR) protection using sophisticated techniques such as memory disclosure exploits [80], custom stack at fixed location [130]. ASLR is a defense technique implemented in modern OS, which randomizes the address of the stack and base address of the libraries and executable in order to defend against memory corruption attacks.

Now, we brief the attack model for ROP and JOP attacks. Fig. 5.1 illustrates the program memory layout for common ROP and JOP attacks. ROP uses gadgets ending in a return instruction [65], while JOP uses gadgets ending in an indirect jump instruction [68]. First, the control structure is corrupted to create the requisite payload. The ROP attack payload consists of return addresses, which targets the gadgets ending in `ret` instruction in the library or victim code. Initially, the stack pointer is moved to the manipulated return address. The execution of `ret` instruction in the gadget moves the stack pointer to the next return address. The stack pointer is used to orchestrate the gadgets in sequence. While in the case of JOP, the dispatcher gadget is used, which acts as virtual program counter

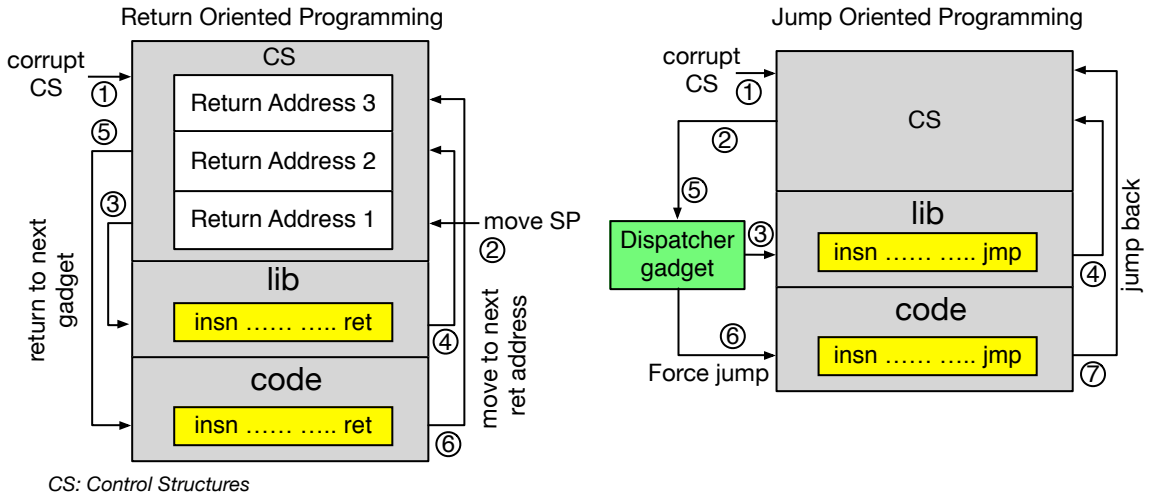


FIGURE 5.1: Program memory of ROP and JOP attacks

(PC), to advance the execution control from one gadget to another. Each gadget contributes certain computations to perform the complete attack.

5.3 Related Works

We discuss the related work in two parts. The first part summarizes the software-based protection approaches, while the second part discusses the more relevant hardware-based architectural techniques. We grouped the relevant defense approaches based on their techniques and implementation platform (software or hardware), as shown in Table 5.1.

5.3.1 Software-based Approaches

The runtime attacks on memory mainly use the buffer overflow technique [59], which still remains the most prevalent attacks till date, and code reuse attack (CRA). Stack smashing is a common type of buffer overflow technique used by attackers to inject their own code in the stack. Several software techniques have been proposed to prevent buffer overflow attacks. StackGuard [59] uses a marker between a function pointer and a return address in the stack, whereas RAD [131]

uses protection code into the prologues and epilogues of function calls to protect the overflow of the return address.

To protect from the code reuse attack, Davi et al. proposed a technique to monitor the repeated small sequences of instructions ending in return statements [86]. Li et al. [132] proposed binary rewriting techniques to replace the return statements. The major limitations of software techniques are the high performance overhead and increase in the code size resulted from the source code or binary modification. Besides, the software approaches have limited coverage of security vulnerabilities, which lead to unknown attacks.

The CFI approach was proposed by Abadi et al. [72] to prevent software attacks during runtime. The main hypothesis is that the CFI enforcement ensures the program execution to follow its predetermined CFG. Although the CFI defense technique is intrinsic and quite promising, it suffers from the limitations of high performance overhead and the increase in code size due to binary modification. To overcome the previous limitations, Zhang et al. [126] proposed Bin-CFI policy. This approach reduces the performance overhead and also addresses the exceptional cases (e.g., multithreading, C++ exception handling). However, it is coarse-grained, as it allows the `ret` instruction to target any address after the `call` instruction, which is prone to sophisticated exploits.

In addition, the software solutions are mainly based on the known instructions at the compilation time, which can be bypassed by the unintended branch instructions. The unintended branch instructions are formulated by pointing the PC to the middle of the multi-byte instructions. For the ISA such as Intel, which has multi-byte instructions, pointing to the middle of instructions leads to the formation of different instructions that are not intended during the compilation time.

5.3.2 Hardware-based Approaches

Of greater relevance to our approach are the micro-architectural techniques. Data Execution Prevention (DEP) was implemented as a hardware memory protection

Table 5.1: Defense approaches against runtime memory attacks

Techniques	Software-based	Hardware-based
Isolation based	StackGuard [59], RAD [131]	DEP [110, 111], XOM [109]
Runtime code integrity		REM [107], DIC [133]
Binary rewriting	Li et al. [132]	
Signature-based	Davi et al. [86]	Kayaalp et al. [134]
Control flow integrity	Abadi et al. [72], Zhang et al. [126]	Arora et al. [106], kBouncer [135], CFIMon [75], Davi et al. [77], BR [76], Das et al. [113]

technique [110, 111] to prevent code injection attacks. DEP does not allow a memory page to be writable and executable at the same time. Several techniques have been proposed to protect from runtime attacks on control flow. XOM architecture uses compartmentalized storage of program code in order to isolate programs running on the same CPU [109]. It requires the program to be encrypted by the vendor and the program is first decrypted when loaded from memory. However, XOM protects only encrypted programs, whereas most of programs and library code available are in the unencrypted form. In [106], Arora et al. proposed a hardware-assisted architecture to monitor control flow at three granularity levels: inter-procedural level with program's static function call graph, intra-procedural level for each branch and jump within a function, and integrity of instructions. However, this technique does not consider indirect branches, which can lead to attacks such as code reuse. Also, it has high performance overhead due to the integrity checking at instruction level. REM [107] architecture monitors the program execution by verifying the integrity of the program at the hash block level. The technique requires modification of ISA and suffers from memory overhead. Kanuparthi et al. [133] proposed Dynamic Integrity Checker (DIC) to protect program integrity at instruction fetch stage of the pipeline. DIC verifies integrity at trace

level with better performance. However, similar to REM, the aforementioned designs are not able to detect attacks that use indirect branches to change the control flow.

Several works leverage hardware features to defend against ROP attacks. Kayaalp et al. [134] gave a signature-based method on the execution patterns. kBouncer [135] uses the last branch recording feature of modern processors to retrieve the most recent indirect branch instructions and validates them using CFI policy. But, the approach is coarse-grained as the `ret` instruction can target any instruction after `call`, whereas `call` and `jmp` instructions are not verified. CFIMon [75] employs the branch tracing store mechanism and performance counters to monitor the control flow. However, it allows indirect jumps to target any instruction, which makes it prone to JOP attacks.

Davi et al. proposed a fine-grained CFI technique using state model and per-function label [77]. The approach has several limitations: it requires ISA modification, compiler extensions and binary rewriting (which increases the code size). In addition to that, the CFI policy will result in false positives due to strict policy allowing the `ret` instruction to target only the instruction following `call` in the active function. For example, in a multithreaded program, a `call` instruction of one thread may be followed by a `ret` instruction of other thread, which will target the return address in an inactive thread; in C++ exception, a `ret` instruction targets caller function instead of return address (explained later in Section 5.4.3). Besides, the policy for indirect jump is based on heuristics with no fundamental rationale.

Branch regulation (BR) [76] is the closest work with our approach. It verifies the target address of branch instructions to monitor the control flow at the execution stage of the pipeline. However, this method increases the code size, as it annotates the binary. Our control flow checking is done at the basic block level, while BR performs at the function level. The `jmp` instruction is allowed to transfer the program execution to any address within the boundary of the same function or at the starting address of the other function. This will result in false positives in the

case, where indirect jump targets middle of the function, e.g., *longjmp()* (shown in Fig. 5.6). Our BB-CFI policy allows indirect `jmp` to target basic block of other function as well, thus it will not result in false positives. To recall, unintended branches are formulated if the PC points to the middle of multi-byte instructions. As the indirect jumps are allowed to target any address inside the same function, the probability of finding the unintended branches increases for a larger function. Hence, BB-CFI limits the scope for attackers to find the unintended branches more strictly than BR. Furthermore, all the hardware approaches do not address the aforesaid exceptional cases.

5.4 Basic block CFI (BB-CFI)

In this work, we propose an approach, named BB-CFI, which enforces a fine-grained control flow execution to protect against buffer overflow and code reuse attacks. In general, control flow instructions (CFINs), such as `call`, `ret` and `jmp`, are used to transfer the program execution to an intended target address (TA). Adversaries attempt to manipulate TA of CFINs to perform attacks. The key idea of our approach is to verify TA of CFIN to enforce the normal control flow of the program during runtime. An unexpected or deviated control flow from the original control flow graph is considered as an attack. An alarm signal (e.g., a hardware interrupt to the OS) is generated to abort the program upon the detection. Depending on ISA, CFINs may include other instructions apart from `call`, `ret` and `jmp`. Such instructions also need to be considered for verifying the control flow. Without loss of generality, we limit our approach to `call`, `jmp` and `ret` as CFIN, where new CFINs can be incorporated in a similar way.

As shown in Fig. 5.2, our approach consists of two steps: 1) offline *profiling* of the program and 2) runtime *control flow checking*. The profiling of the program is dynamic, i.e., during the program execution. During the offline profiling stage, the program is first executed with a set of inputs. It is then disassembled into its CFG. Fig. 5.4 illustrates the CFG of functions, where each node represents a

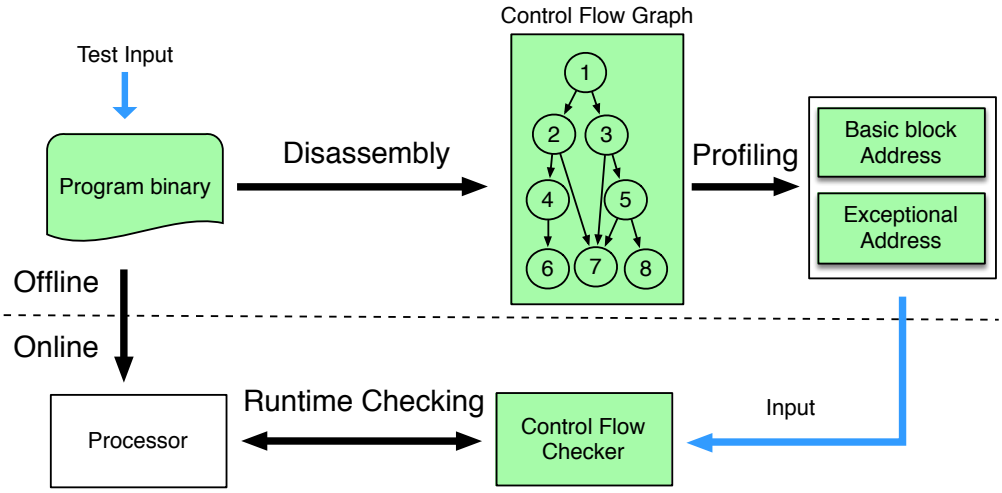


FIGURE 5.2: Overall approach of BB-CFI

basic block. The profiler obtains the basic block information and the exceptional handler addresses from the CFG. These data are recorded and provided to the control flow checker. At runtime, the control flow checker verifies the program execution on the processor using these reference data.

Now we discuss the two steps of BB-CFI approach (profiling and control flow checking) in detail. Then, we explain the exceptional cases, which may cause false errors in the control flow verification. Finally, we propose the extended rules to handle them in Section 5.4.3.

5.4.1 Profiling

Profiling plays a critical role in our method. We aim to retrieve all the necessary control flow information to validate the runtime execution of the program. We assume that the profiling process as well as the program binary used in profiling are trusted. Any modifications in the program binary before profiling cannot be detected by our method, because we trust the control flow information generated by profiling. Similar to other security approaches [107, 108, 133], we consider the profiling information as the trusted computing base of our approach. In Fig. 5.2, the trusted steps are highlighted (in green color). We advocate that profiling has to be done by the developers or the trusted third party who are familiar with the application. Similar profiling technique has been used by previous approaches, e.g.,

```
0x08048443:   push   ebp
0x08048444:   mov    ebp,esp
0x08048446:   push   edi
0x08048447:   push   esi
0x08048448:   mov    esi,edx
0x0804844a:   push   ebx
0x0804844b:   mov    ebx,eax
0x08048443:   jmp    0x08048448
0x0804844d:   sub    esp,0x6c
0x08048450:   mov    edi, ebx
0x08048457:   mov    DWORD PTR [ebp-0x60],ecx
0x0804845a:   call   0x080923b0 <__wctrans>
```

FIGURE 5.3: Code snippet showing basic block address

for runtime integrity verification [107, 108, 133], malware detection [53]; REM [107] profiles the hash value of a basic block code during its first run on the system; DIC [133] and [108] profile the hash value for a trace (i.e., a combination of basic blocks in a program path); similarly, [53] obtains the behavioral profile – consisting of information such as memory allocations, memory reads/writes, system calls, control transfers in a basic block – during offline. We acknowledge that capturing the complete CFG of the program (i.e., achieving 100% program coverage) by disassembling the binary is a challenging task. Several research works already focus in this direction [136]. In our approach, we only aim to capture the program paths that cover the main functionality of the program. Profiling, being a separate offline process, can be improved by using advanced techniques (e.g., symbolic execution [136]) to increase the program coverage. Since this is not the key focus of our work, we use the existing profiling tools. To cover the main functionality of the program, it also requires to consider different test inputs, input/output interface and user interactions during program execution. With a good understanding of the program and systematic profiling, it is possible to achieve a high coverage of CFG and hence, a low false positive rate, as we show in the evaluation (in Section 5.6.2.4).

As stated earlier, the profiling of the program is done to obtain the basic block information. The program is first executed with the test inputs to generate the instruction traces, as shown in Fig. 5.3. From the instruction traces, a list of

basic block addresses are retrieved by grouping the instruction sequences into blocks such that each block has a single entry point and a single exit point [137]. For example, in Fig. 5.3, there are four basic blocks at addresses: *0x08048443, 0x08048448, 0x0804844d and 0x080923b0*. Note that a basic block address list also contains constant code pointers, which are target addresses of CFINs, e.g., `jmp 0x08048448`. The address of the first basic block of a function is identified by disassembling the binary. Our method extracts BB information, consisting of starting address of BB (BB_ADDR) represented by 32-bits and 1-bit information representing whether it is the first BB of the function. This information is stored as the reference data for verifying the control flow at runtime.

The profiling is done on the binary executable of the software in our architecture. Thus, our method does not require the source code. This is an advantage of our method, as the source code of the legacy software may not be available. The profiling is performed using different sets of inputs, which are required to execute the program. During profiling, for each set of inputs, the BB information is collected assuming it covers a part of the complete CFG of the program. In order to increase the coverage, the program is executed multiple times with different sets of valid inputs, keeping in mind of all the possible inputs that can be given at the runtime. The collected data is stored in the FPGA block memory. Main memory can also be used to store the BB information, if the FPGA memory is not sufficient (as shown in Fig. 5.8). Note that we only use the protected region in main memory (along with OS kernel) to store the BB information, similar to the previous approaches (e.g., dynamic integrity checking [133]). The kernel region in main memory is not accessible to the user programs [138], and therefore, the adversaries cannot modify it.

The attempt to use the malicious inputs at runtime might lead to a different control flow. Thus, our control flow checking technique can detect the abnormal deviation in control flow by verifying with the reference data. In the case of lack of complete input coverage, BBs information might not be sufficient to verify the correct control flow and normal control flow might be considered as an attack. As

a future work, we can use advanced program analysis techniques, e.g., symbolic execution [136], to have a complete coverage of the program during the profiling.

During profiling, we also extract exception handler landing pad addresses (EH) for the program. Once the exception, such as C++ exception, occurs during the program execution, the stack unwinding process transfers the execution control to the landing pad address. These addresses are extracted from the binary executable, which is detailed later in Section 5.4.3.

In a nutshell, we extract the following information during profiling:

- Basic block address (BB_ADDR)
- Address of the first basic block of a function
- Exception handler landing pad address (EH)

5.4.2 Control Flow Checking

In general, attacks such as stack smashing and ROP use buffer overflow to overwrite the return address stored in the stack and transfer the control to an arbitrary address. By checking the return address, most of the previously proposed techniques are able to detect these attacks. In response to these defensive techniques, JOP attacks were exploited as a new type of attacks with little variation. The JOP attacks are performed by manipulating the target address of the indirect jumps, instead of return address, to deviate the program flow. In order to defend JOP attacks, our method verifies the TA of indirect jumps to ensure normal execution of a program. CFINs are responsible for transferring the execution control. Each function in the program may contain one or more BBs. The program execution transfers from one BB to another within the same function or to a different function. As there is a single entry point for each BB, the CFIN can only transfer the program execution to the starting address of the next BB, which eases the verification of the control flow. For example in Fig. 5.4, the control flow from $BB1\_2$, i.e., the second block of *func1*, can only be directed to the `add` instruction, which is the

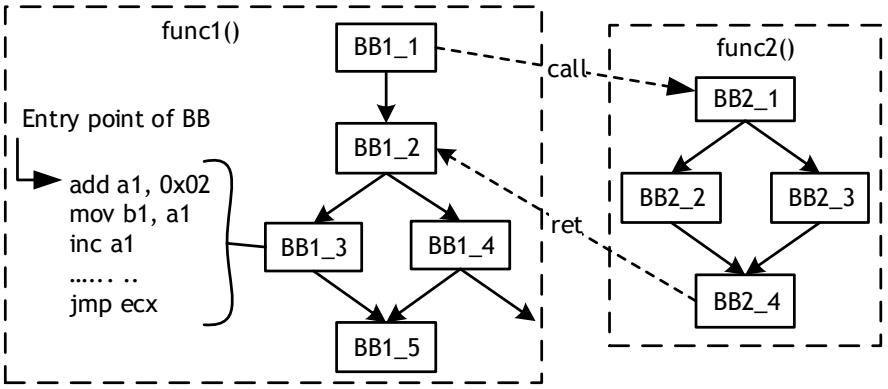


FIGURE 5.4: Control flow graph for functions

first instruction of $BB1\_3$. However, the execution control cannot be transferred to the middle of the BB. We use this fundamental concept in addition to few rules (mentioned below), to verify the control flow. Our method provides fine-grained control flow integrity at BB level, named BB-CFI. Aforesaid attacks, ROP and JOP, deviate the control flow to the gadgets, which are of a few instructions and are not at the starting of the BB. Since our method allows the control to be transferred only to the starting address of the BB, such attacks can be detected. The normal program execution can be guaranteed if the execution follows the CFG of the program decided at the compilation time. If there is an attack diverting the program to a different address at runtime, it can be detected by comparing the target address of CFINs with the pre-stored profiling data of BB_ADDR. We propose the following verification rules for each CFIN:

- `call`: The target address of a function call must be the first instruction of the called function. In order to verify the TA of the `call` instruction, our method checks if the TA is the BB_ADDR and also the first BB of a function. For example in Fig. 5.4, the `call` instruction of $BB1\_1$ in *func1()* targets at $BB2\_1$, which is the first block of *func2()*.

- `jmp`: Jump instruction can transfer the control to BB_ADDR, such as `jmp` from $BB1\_3$ to $BB1\_5$. This is similar to the `call` instruction except that jump is allowed to target at any BB inside the same or a different function. It is worth mentioning that some CFI policies [76] allow jumps to target inside

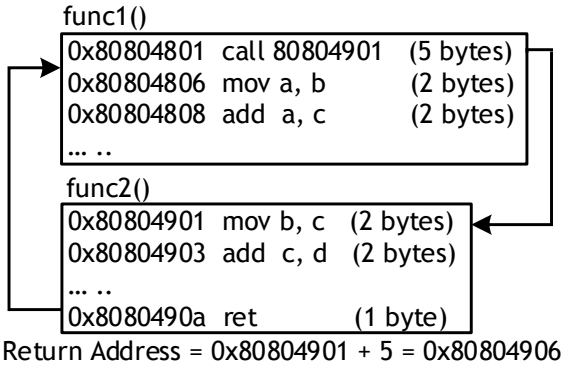


FIGURE 5.5: Return address calculation at `call` instruction

the same function boundary or at the starting address of other function. Such policies will result in false error in cases where jump targets at the middle of the function. This is shown in Fig. 5.6 and explained in Section 5.4.3.

- `ret`: We allow `ret` instruction to return to the instruction following the corresponding `call` instruction, called as `call-ret` pairing. In Fig. 5.4, the `ret` instruction transfers the execution to the first instruction in $BB1\_2$ in *func1()*, as this is the instruction following `call` in $BB1\_1$ (from where *func2()* was called).

  We compute the return address at every `call` instruction at the runtime and store it in the return address stack (RAS), which acts as a *last-in-first-out* buffer. For the verification, the target address of `ret` instruction is compared against the last entry of RAS. Then the corresponding return address is evicted from RAS. The return address is computed as the sum of the address of `call` instruction and the length of `call` instruction. We illustrate using the example in Fig. 5.5. At the `call` instruction, the return address is computed and verified after the execution of `ret` instruction. The control execution returns to address *0x80804806*, which is the instruction following the `call`.

TABLE 5.2: Normal and exceptional cases of BB-CFI

Normal cases	Exceptional cases
1. `call` targets function entry	1. `jmp` targets return address
2. `jmp` targets basic block entry	2. `ret` targets caller function
3. `ret` targets the instruction following corresponding `call`	3. `call-ret` pairing is not followed 4. `ret` is used as jump

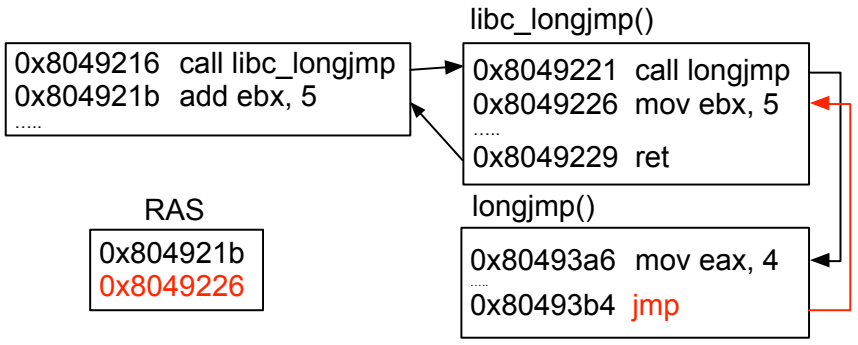


FIGURE 5.6: Exception 1: `jmp` instruction targets return address

5.4.3 Exception Handling

1) Exceptional cases

In complex binaries, there are some exceptions to the normal behavior of `call`, `ret` and `jmp`. For instance, `call-ret` pair is replaced by `call-jmp` pair. Thus, most of the previously proposed CFI methods, which strictly follow `call-ret` pairing, result in a false positives for such binaries even for the normal control flow of the program. In Table 5.2, we list the normal and exceptional cases.

In this section, first we explain the exceptional behavior of CFINs. We show how the earlier proposed CFI techniques result in false positives for these cases, i.e., misidentifying the normal behavior of the program as an attack. Then we propose extended rules to our original BB-CFI to support these exceptional cases.

Case 1: `jmp` instruction targets return address

Functions such as *longjmp()* use an indirect `jmp` instruction instead of `ret` to target

return address. As shown in the Fig. 5.6, `jmp` instruction is used for returning to *libc_longjmp()* instead of the `ret` instruction. CFI techniques [76, 77] allow `jmp` instruction to target active function (i.e., inside the boundaries of the same function) or at the entry of other functions. Such CFI policy will result in a false positives in case of *longjmp()* because the `jmp` targets at the middle of the *libc_longjmp()*.

In our method, such `jmp` instruction verification does not show error as it targets the BB_ADDR. However, `call-ret` pairing provided by RAS is affected, which later results in a false positives for the verification of the subsequent `ret` instruction. As depicted in Fig. 5.6, the return addresses corresponding to two `call` instructions are stored in RAS. Since `jmp` is used instead of `ret` (at `0x80493b4`) in *longjmp()*, `call-ret` pairing in RAS is affected. This is reflected during the verification of subsequent `ret` (at `0x8049229`), where the expected target address (`0x8049226`) (since it is the last entry in RAS) does not match the actual target (`0x804921b`). In such cases, the strict `call-ret` pairing fails the validation, resulting in false positives.

Case 2: `ret` instruction targets caller function instead of return address
For example, in C++ exception handling, the `ret` instruction returns at the end of *try* block, instead of the return address. As illustrated in Fig. 5.7, after the exception occurs in *funB()*, stack unwinding process begins releasing the stack until it reaches a return address that resides in a *try* block. Finally, the execution control is transferred to the exception handlers at the end of the *try* block in *main()*, rather than to the first statement following the function call *funB()* in *funA()*. Since our method expects the `ret` instruction to transfer the execution control to the last return address in RAS, i.e., to the statement following *funB()* in *funA()*, a false error is shown during verification. In this case, previous CFI implementations would fail too, as `ret` instructions only target the instruction following `call`.

Case 3: `call-ret` instructions appears in different order
For example, in a multithreaded program, `ret` instruction of one thread may

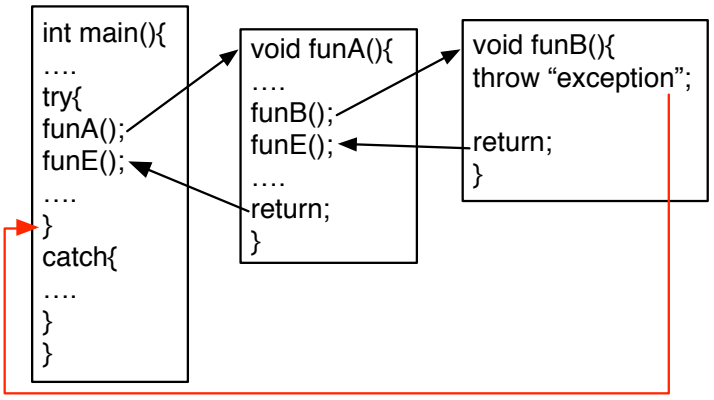


FIGURE 5.7: Exception 2: `ret` instruction targets caller function

appear after `call` instruction of other thread. This is because threads work concurrently and each thread has a separate stack. Since our method has only one RAS and allows `ret` instruction to target at the last entry in RAS (i.e., instruction following last `call` instruction), for a proper functioning `call-ret` pair of a thread must appear in order. However, this is not the case in multithreaded programs, which causes false error during the verification of `ret` instruction. Previous CFI implementations following `call-ret` pairing would also have a similar error, in lack of a separate `call-ret` stack for each thread.

Case 4: `ret` instruction is used as jump

A `ret` instruction is sometimes replaced by `pop` and indirect `jmp` instructions by the compiler. Our method has no error during the verification of `jmp` instruction. Nonetheless, subsequent `ret` instruction verification results in a false error owing to the fact that `call-ret` pairing in RAS is affected, similar to Case 1.

2) Extended rules

We propose extended rules to adapt to the exceptional cases:

Rule 1: Indirect jumps can target return address

To address the exceptional Cases 1 and 4, we allow indirect `jmp` to target a return address in RAS. Corresponding return address is removed from the RAS after verification, which allows proper verification of subsequent `ret` instruction.

Table 5.3: BB-CFI policy: Target Address for CFINs

CFI	Original BB-CFI	Extended rules
`call`	First BB_ADDR	
`ret`	Return Address	EH
`jmp`	BB_ADDR	Return Address
`call-ret`	In-order	Order-independent

Rule 2: `ret` instructions can target Exception handler landing pad address (EH)

In order to adapt to C++ exception (Case 2), we allow `ret` instruction to target EH. The EH addresses can be obtained from the ELF binary using the information from the two sections *.eh_frame* and *.gcc_except_table*, which are written in DWARF debugging format. These two sections can be parsed to retrieve the base address and offset in order to compute the EH addresses [139, 140].

Rule 3: `ret` instructions can target any address in RAS

For a multithreaded program (Case 3), where `ret` instruction of one thread may appear after a `call` instruction of other thread, return addresses must be represented uniquely for each thread in a RAS or a separate RAS has to be assigned for each thread. Since our approach works at the hardware level (which is below the OS layer), thread context switch cannot be determined. Therefore, we relax our CFI policy by allowing the `ret` instruction to target any address in RAS in an independent order, rather than allowing to target only the last RAS entry. Thus, different threads share the same RAS. However, we emphasize that strict `call-ret` pairing can be followed for a multithreaded program in our approach, if each RAS entry is identified by a thread identifier. The threads can be differentiated by getting the OS support during thread context change.

Table 5.3 presents the targets for CFINs allowed by BB-CFI and additional targets allowed by the extended rules. As stated, `ret` instructions can additionally target EH, while `jmp` can also target a return address in RAS, as compared to the original policy. These extended rules do not provide a scope for attackers to exploit them. Rather they (Rules 1 and 3) are proposed for normal functioning of

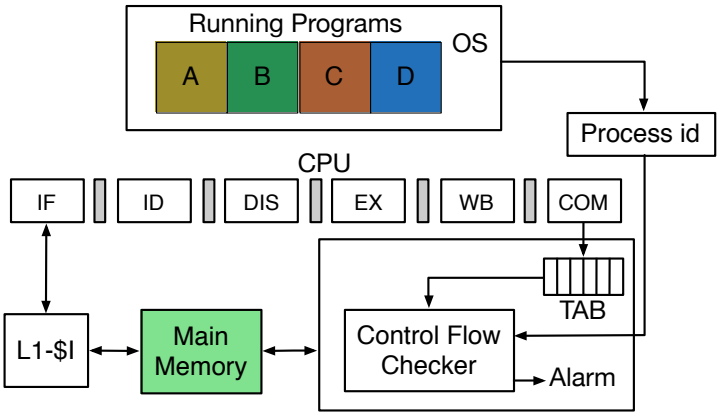


FIGURE 5.8: Micro-architecture design of CFC

original BB-CFI, as most of the exceptions affect `call-ret` pairs in RAS. Rule 1 allows jumps to target at return addresses, which are also BB_ADDR, and hence, remains consistent with the original BB-CFI rule for jumps. Rule 3 allows `ret` to target already computed return address, with only change in the order. The only flexibility an attacker gets here is the change in the order of return addresses, which is not sufficient to perform any attack. However, the attackers can use Rule 2, provided that they manipulate EH in the binary file. Since we assume the binary file is intact and trustworthy during profiling while we extract the EH, any such modification after time-of-compilation and before time-of-use can still be detected by our method.

5.5 Micro-architecture design

Our control flow checking technique consists of Control Flow Checker (CFC) to perform the runtime checking and the Target Address Buffer (TAB) to buffer the addresses which need to be validated by CFC, as shown in Fig. 5.8. In this section, first we explain the design considerations of CFC. Second, we detail its micro-architecture design and explain how we address these design considerations. Third, we discuss the advantages and limitations of our proposed architecture.

CFC involves the following design considerations:

1. *Tight integration with the existing CPU core*

   The CFC should be able to access the TA of CFINs during program execution for its verification. This requires a tight integration of CFC with the existing CPU core.

2. *Low performance overhead*

   The performance overhead on the processor offered by the security logic needs to be low. The large delay due to the verification logic can cause the processor to stall for a longer time, finally resulting in a high performance overhead, which is undesirable. For this reason, the CFC verification logic should offer minimal overhead.

3. *Real-time detection*

   The CFC should be able to operate faster so that the attacks can be detected in real time and the system can be protected from the major damage.

4. *Multitasking support*

   The design should facilitate the support of multitasking operation. This is challenging because the CFC operates in the hardware level, i.e., below the OS layer, at which the process identifier/context switching of process cannot be identified directly.

5.5.1 Micro-architecture Design of CFC

In this work, we implement the CFC design in FPGA as the proof of concept. As compared to ASIC, FPGA implementation offers lower cost, fast prototyping and flexibility to update the verification logic if it is compromised. The proposed design can also be implemented in ASIC platform based on the foundation of this work. Our security approach can be updated. Profiling process can be independently updated as it is a separate offline process. The verification logic can be updated to incorporate the additional rules – to enhance the CFI, to reduce false positives or to increase the performance. The verification logic can be implemented either in ASIC or FPGA platform. In comparison to FPGA, ASIC has less overhead in terms

of area and performance. However, the core logic cannot be modified in ASIC. Whereas, the FPGA provides flexibility to update the core logic during runtime using its reconfigurability feature. Thus, if the verification logic is implemented in FPGA, our approach can support complete security update, including profiling and verification logic during runtime.

We assume System on Chip (SoC) architecture for our design, consisting of FPGA and CPU on the same chip, as shown in Fig. 5.8. Hence, no hardware attack is feasible on the bus connecting them. We also assume that FPGA logic can be updated using bitstream encryption techniques, from the main memory in the trusted environment. Bitstreams can be stored along with the operating systems (OS) in main memory, as this region of the memory is secured and not accessible to the application programs. At the design time of the SoC, the FPGA chip for CFC implementation is connected between the commit stage of the CPU and the main memory using separate interconnects as shown in Fig. 5.8. We integrate our CFC module at the commit stage of the CPU pipeline for two reasons: first, the target address of the indirect CFINs are computed only at the execution stage of the pipeline, and second, our control flow checking methodology requires instructions to be in program order, which in some processor occurs only at the commit stage. This explains the first design consideration of CFC, as stated earlier.

At the commit stage of the pipeline, we collect TA for each CFIN. This can be known from the PC value, as PC points to the TA after the execution of CFIN. This information is appended with the CFIN type, which represents either `call`, `ret` or `jmp`. Return address of the function (which is computed as explained in Section 5.4.2) is also appended if the CFIN is `call` instruction. Two bits are required to represent the three types of CFINs. The return address and TA are 32-bit each. Hence, it requires 66 bits if the CFIN type is `call` and 34 bits if the CFIN type is `ret` or `jmp`. This information is stored in the embedded memory block on FPGA for the purpose of runtime verification.

The internal architecture of CFC is shown in Fig. 5.9. CFC gets the input from TAB. TAB holds CFIN information retrieved from the commit stage of the CPU

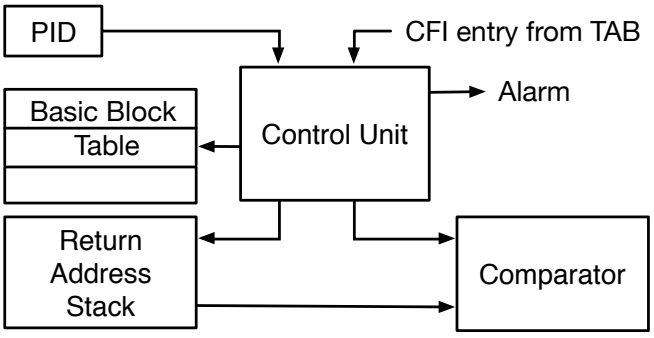


FIGURE 5.9: Internal design of CFC

pipeline. The TAB buffer is used to reduce the performance overhead of CPU caused by the verification. The processor runs comparatively at a higher speed than the FPGA. Meanwhile, CFC takes few cycles to perform the validation. If the CFIN arrives faster in the pipeline than the time taken by CFC to validate the last entry, the CPU pipeline has to be stalled until CFC completes its task. As a consequence, there will be significant overhead in the performance of the system. In order to mitigate this overhead, we use TAB to buffer the generated CFIN entries and allow CPU to continue execution without stall. This describes our second design consideration.

The other design consideration is the support of multitasking operation, which is achieved by tracing the control register CR3. As depicted in Fig. 5.8, the process ID is obtained by tracing CR3 register. OS assigns a unique page directory for each process. Intel x86 architecture stores the physical address of base of page directory for the current running process in control register CR3. Upon context switching, the entry in CR3 is overwritten by the physical address for the new process. Thus, CR3 register can be monitored for multitasking support. Similar technique has been used in [117].

CFC is composed of Basic Block Table (BBT), Control Unit (CU), and Return Address Stack (RAS). BBT is used to store 33-bit BB information obtained from profiling as the reference data and is implemented in FPGA block memory. CFC also gets process ID information by tracing the control register CR3, as stated earlier. BBT is implemented using Content-addressable memory (CAM), which is designed to search the TA in the entire BBT in a single operation cycle and

thus speed up the searching process. CU fetches CFIN entry from the TAB and validates the TA against BBT according to BB-CFI policy, as explained in Section 5.4.

CU generates the immediate alarm signal to abort the program execution if the TA does not follow the rules, indicating possible anomaly in the control flow of the program. This alarm can be hardware interrupt signal to the OS in order to abort the program. If the CFIN is a `call` instruction, CU stores the 32-bit return address into the RAS, which is used for verification of corresponding return address of the function. RAS stores the return address corresponding to each `call` instruction. Finally, the comparator is used to validate the TA of the `ret` instruction with the address in RAS, as stated by the extended rules (Rule 3) in Section 5.4.3. The `ret` instruction is also allowed to return to the EH, if it does not match return address in RAS (Rule 2). To recall, EH obtained from profiling is also stored in BBT. For an indirect `jmp` instruction, TA is first compared with the return address in RAS (Rule 1), and in the case of no match, it is searched in the BB_ADDR list.

BB information for the applications can be stored in BBT, by allocating the memory region for each process. Upon process context change, BB information can be fetched from allocated region of BBT. In a multitasking environment, BBT size may not be sufficient to store the profiling data of all the programs. Thus, main memory can be used to store the BB information. The profiling data of a program can be fetched by OS and stored in BBT before its execution. RAS can be shared among processes for storing the return address. Unique process identifier has to be appended with the return address in RAS. Since CFC implements small logic, it can fit into SoC design, which has small FPGA fabric with the required memory. It will not slow down the commit stage and thus, will not affect the pipeline speed.

5.5.2 Security Analysis and Limitations

Our proposed architecture can defend against the stack smashing and code reuse attacks. The stack smashing and the ROP attacks are based on overwriting of the return address, which are defended by the `call-ret` pairing verification. JOP attacks manipulate the targets of indirect `jmp` instruction, which is also verified by our method. Therefore, all the common attacks, which deviate the normal control flow, can be defended by our method. In addition, the formulation of unintended branch instruction, which is created by pointing the PC value to the middle of multi-byte instruction, is restricted by our method because the execution transfer is validated for each CFIN. In general, to perform ROP and JOP attacks, small sequences of instructions ending in indirect CFINs are required, that are mostly inside the BB rather than at BB_ADDR. Since we allow execution control to transfer only to the BB_ADDR, our method can easily evade these attacks. This is demonstrated on our custom shellcode attacks and RIPE benchmarks [128], explained in Section 5.6.2. CFC can also detect any modification in binary between time-of-compilation and time-of-use, if it violates our control flow policy. For instance, if `call 0x804323` is modified to `call 0x802121`, our method checks if `0x802121` is first BB_ADDR. And it considers as an attack in false condition. Furthermore, our method has no false positives in the exceptional cases (Section 5.4.3) as they are considered by the extended rules.

However, our approach also has some limitations. BB-CFI will have false alarms in case of incomplete profiling of normal program execution. This is because, it will lack the basic block information for the valid program paths that were not covered during profiling. Thus, it requires rigorous profiling using all the valid inputs to maximize the coverage of the normal program paths. CFC can be circumvented if the code has been intruded before entering the processor boundary (such as in the main memory) and the intruded code conforms to our BB-CFI policy. For example, our method cannot detect binary modification, if the instructions are modified inside the BB without modifying CFINs, because it may still follow our BB-CFI policy. As discussed, CFC can detect the common code intrusion that

uses buffer overflow in the stack region. While the code reuse attacks depend on gadgets, there may be few basic blocks, which might be useful for gadgets. This is shown by our results on gadgets elimination in Section 5.6. However, first the gadgets need to be sufficient to perform an attack. On top of that, using these gadgets to perform a meaningful computation requires overwriting of the return address, which is prohibited by the `call-ret` pair in our method. BB-CFI approach may fail to detect an advanced attack, if it uses basic blocks as gadgets and still conforms to our `call-ret` pairing rule. To detect such attack, our approach can be enriched with the heuristic model comprising of the detailed information (e.g., the number of instructions in the gadgets, ratio of consecutive return and call instructions, indirect branches).

Since our method uses control flow information obtained by offline profiling of binaries, it does not support dynamic codes, e.g., just-in-time compilation, self-modifying binaries. Most of the previous CFI implementations also share this limitation. Our method relies on basic block addresses obtained during profiling. Hence, it does not support ASLR. ASLR is an OS based randomization technique to defend buffer overflow and return-to-libc attacks. However, it cannot defend code reuse attacks. In comparison, our method is based on CFI technique and protects from code injection based buffer overflow, return-to-libc and more advanced ROP and JOP attacks. Hence, our method is parallel but more advanced than ASLR.

Our architecture requires that CFG information must be stored in the protected region of the main memory along with OS kernel. Since the kernel region in main memory is not accessible to the user programs [138], in general, the attackers cannot modify the CFG information. However, if the protected memory region is compromised in the worst case, the CFG information can be modified. Such attacks can also lead to the compromise of the OS, which can potentially defeat most of the security protections. However, they are difficult to perform.

Our approach aims on monitoring the control flow to detect these attacks, although it cannot prevent them. The other limitation of our architecture is the

late detection of attack by a few cycles; the reason is due to the slow speed of FPGA as compared to the processor. Nevertheless, the alarm will be triggered before any major damage can be performed by the attack during this small duration of tens of processor clock cycles. In order to improve the performance of CFC, one promising way is to employ parallel checkers to speed up the validation. However, the BBT and RAS also need to be adjusted to allow multi-access at the same time. We leave this as one future work in this line of research.

5.6 Evaluation

5.6.1 Experimental Setup

The framework for CFC, which is targeted for SoC platform, was built using a processor simulator and an FPGA simulation environment. We modified Multi2sim [141], a cycle-accurate, out-of-order simulator (configured for 32-bit, x86 processor), to verify the functionality of CFC. Xilinx ISE 14.1 was used for simulation and implementation of CFC architecture. After verifying the functionality, we implemented the CFC architecture on FPGA device of Virtex-5 LX110T and obtained corresponding area and delay results. The parameters (operating frequency and number of cycles) obtained for the CFC were then plugged into the processor simulator to perform system-level simulation. Power estimation of the CFC was done using XPower analyzer tool. We used Synopsis Design Compiler tool to evaluate the area and power for CFC on ASIC platform at 65 nm technology.

To evaluate our BB-CFI defense, we used Runtime Intrusion Prevention Evaluator (RIPE) [128] test suite, which consists of 850 buffer-overflow attacks. ROPGadget-v4.0.3 tool [129], which scans the binary to search gadgets useful to facilitate CRA exploitation, was employed to estimate the gadgets elimination. We selected SPEC CPU2006 [142], MiBench [143], BioBench [144] and Stream [145] benchmarks to evaluate the performance of our approach. We obtained the compiled SPEC benchmark from Multi2sim website [146]. GNU gcc version 4.6.3 was used

Table 5.4: RIPE Benchmark evaluation

	DEP disabled	DEP enabled
Total	503	131
Failed	20	20
Successful	483	111
BB-CFI	483 (483)	111 (111)
Bin-CFI	430 (520)	50 (140)

for the compilation of MiBench, BioBench and Stream benchmarks at O3 optimization level. To profile the benchmarks, "exp-bbv", a basic block vector tool provided by Valgrind [147] is utilized. The simulation results were obtained for the representative 100 million instructions for large benchmarks, such as SPEC, BioBench and Stream. For smaller benchmark such as MiBench, the results were obtained for complete benchmarks. *Note that Virtex-5 is used only for demonstration purpose. Real design of CFC needs to be implemented on FPGA fabric, integrated at commit stage of CPU, which has much smaller footprint compared to the large area of Virtex-5.*

5.6.2 Experimental Results

5.6.2.1 Evaluation on RIPE benchmark attacks

RIPE performs exploits using techniques – code injection, return-into-libc, ROP – on stack, heap, BSS and data segment. It bypasses ASLR protection as it calculates the target address and offsets at runtime, and therefore the information is available after randomization. We tested our approach by enabling/disabling DEP, which can be bypassed as mentioned in our threat model (Section 5.2). As shown in Table 5.4, out of 850 exploits in RIPE, 503 attacks were successful on Ubuntu-12.04 OS with DEP disabled. However, only 483 attacks were successful on our processor simulator. For 20 attacks, OS terminated the simulator immediately and the simulation was incomplete. All the 483 attacks were successfully detected

Table 5.5: Gadgets elimination

Benchmarks	Total Gadgets	Allowed Gadgets	BB-CFI (%)	Bin-CFI (%)
djpeg	11787	54	99.541	-
blowfish	10604	34	99.679	-
susan.corners	10953	42	99.616	-
401.bzip2	5618	38	99.323	93.33
403.gcc	25947	360	98.612	91.42
462.libquantum	5687	36	99.367	86.36
483.xalancbmk	42250	328	99.223	-
006.phylip	11171	53	99.525	-
stream	10637	48	99.548	-
Average			99.381	90.37

by our BB-CFI approach. For DEP enabled case, all the 111 attacks, that were successful on simulator, were detected by our BB-CFI approach. Most of these attacks – code injection, return-into-libc, ROP – overwrite return address, which is verified by our strict BB-CFI policy (as explained in Section 5.4) and hence, can be detected. Some attacks use `jmp` instruction (e.g., *longjmp* buffer), function pointer to redirect the control flow to the gadgets, which are not the BB_ADDR. Since our method allows `jmp` instruction to target only the BB_ADDR, these attacks get detected. Bin-CFI cannot detect 90 function pointer overwrite attacks, whereas our method can detect all these attacks. This is because our approach uses basic block information, obtained by profiling the normal behavior of the program, to detect any deviation from the normal control flow at runtime. To clarify, 520 attacks were successful when DEP was disabled, while 140 attacks were successful when DEP was enabled, in their environmental setup [126]. In Table 5.4, the number in parenthesis indicates total attacks.

5.6.2.2 Gadgets elimination

To observe the effectiveness of our approach against CRA (ROP and JOP), we evaluate the reduction in number of gadgets offered by BB-CFI. To recall, our method allows `call` instruction to target the first BB_ADDR of a function, `ret` to target the immediate instruction after `call` and `jmp` to target BB_ADDR. Since our method allows the indirect CFINs to target BB only, we consider that any BB, which is a gadget, can be exploited to perform a CRA. This means that any BB_ADDR matching with the gadgets' starting address will be allowed by our method. First, we use popular ROPGadget-v4.0.3 tool [129] to scan the binary and search useful gadgets for CRA exploitation. Then, we use our customized code to evaluate how many of these gadgets are allowed by BB-CFI. Table 5.5 shows our gadgets evaluation for different benchmarks. We also compare our results with the Bin-CFI method for common SPEC programs that are presented in [126]. As shown by the result, BB-CFI effectively enforces the fine-grained CFI at BB level and reduces the gadgets by >99% on average, whereas Bin-CFI reduces them by 90% on average.

5.6.2.3 Average indirect control flow reduction (AIR)

In general, the adversary manipulates the target address of indirect CFINs to hijack the normal control flow of the program while performing CRAs. Therefore, the strength of the CFI approach can be measured by the reduction in possible target addresses for the indirect CFINs. To this end, Zhang et al. [126] proposed average indirect target reduction (AIR) metric to quantify the total reduction of indirect targets, given by Equation 5.1. Intuitively, the AIR measures the reduction in the possible attack opportunities for performing CRAs.

$$\text{AIR} = 1/n \sum_{j=1}^{n} (1 - |T_j|/S) \tag{5.1}$$

where, n is the number of indirect CFINs, T_j is the possible target addresses and S is the size of binary code.

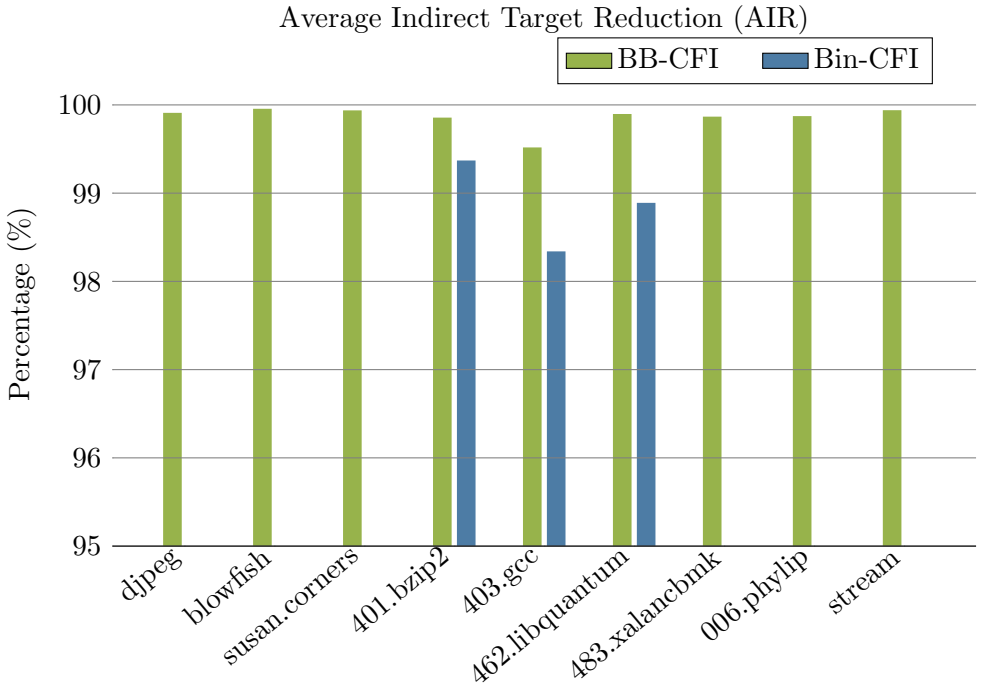


FIGURE 5.10: Evaluation of AIR on benchmarks

In BB-CFI approach, all the `call` instructions target the same set of addresses (i.e., beginning of functions), all the `jmp` instructions target starting address of basic blocks (i.e., BB_ADDR), whereas `ret` instructions target the instruction following the corresponding `call`. Therefore, the number of target addresses for `call` is given by the number of functions, whereas for `jmp` by the number of basic blocks. And for `ret` instruction, it is always 1. This simplifies the Equation 5.1 to:

$$\text{AIR} = (1 - (|T_{call}| + |T_{jmp}| + 1)/3S) \tag{5.2}$$

where, T_{call} and T_{jmp} denote the number of functions and basic blocks in the program, respectively.

Fig. 5.10 depicts our estimation of AIR metric for the benchmarks using Equation 5.2. The average reduction of indirect targets for our method is >99%, which is relatively higher than Bin-CFI method (98%). As compared to Bin-CFI, our method has a marginal increment in AIR, but has higher elimination of gadgets (shown in Table 5.5). This is because that we restrict all the indirect transfers to

Table 5.6: Functional verification of BB-CFI policy

Benchmarks	#call	#ret	#jmp	Total CFIN	Match(%)	Ins. (x10^6)
djpeg	8201	8193	38416	54810	100	6.7
blowfish	999419	999411	63222	2062052	100	65.29
susan.corners	1178	1171	1672	4021	100	1.12
401.bzip2	615809	615800	1099844	2331453	100	100
403.gcc	1690520	1690508	1716260	5097288	100	100
462.libquantum	8002	7993	204071	220066	100	100
483.xalancbmk	2251822	2251800	1294823	5798445	100	100
006.phylip	2114952	2114947	707720	4937619	100	100
stream	278	275	188	741	100	100

the BB_ADDR, whereas their method allows returns and indirect jumps to target any address after the function call. On top of it, our method can detect all the RIPE attacks, whereas Bin-CFI cannot detect 90 attacks.

5.6.2.4 Evaluation of CFI policy

Table 5.6 shows the verification results of `call`, `ret` and `jmp` instructions for different benchmarks obtained from simulation. The simulation results show a matching of 100% of CFIN (with no false error) as per our control flow checking rule for the benchmarks.

In order to verify the extended rules, we modified the benchmarks to introduce the exceptional cases (i.e., multithreading, C++ exception, *longjump()*), as they were not present in the original programs. Our experiment shows that the extended rules do not have any false positive. This is because these rules do not modify our BB-CFI policy but only relax it in order to support the exceptions. However, the extended Rule 2 (Section 5.4.3) increases the reference data, as it allows returns to target at the exception handler landing pad address (EH). The number of such EH addresses depends on the exceptions (*try-catch* block and default exceptions) that

Table 5.7: Memory requirement for Basic Block Table

Benchmarks	#Basic Blocks	Memory size (KB)
djpeg	2031	9
blowfish	861	4
susan.corners	1283	6
401.bzip2	2210	9
403.gcc	51767	209
462.libquantum	1491	6
483.xalancbmk	24325	98
006.phylip	2891	12
stream	1218	5

have been handled in the program. We discuss the memory overhead introduced due to EH addresses in Section 5.6.2.6.

5.6.2.5 Evaluation on shellcode attacks

Since there is no benchmark for JOP attacks and advanced ROP attacks, we wrote customized *shellcode* attack programs to launch *shell* using all three techniques: stack smashing, ROP and JOP. We performed these attacks on the simulation environment with our security module BB-CFI on board. The CFC was able to detect all the attacks successfully. We clarify that these are not real world attacks. Nevertheless, they still follow the general notion of stack smashing, ROP and JOP attacks and are used here for the proof of concept.

5.6.2.6 Hardware evaluation

The memory requirement for storing the BB information varies as per the program. BB information requires 33-bits. Table 5.7 shows the memory requirement for different benchmarks. The results show that our method has relatively lower memory overhead than REM [107] and DIC techniques [133], while they were performing integrity checking. The largest benchmark 403.gcc requires around 209

Table 5.8: CFC resource utilization

FPGA	No. of used Flip flops	50 (<1%)
	No. of used LUTs	86 (<1%)
	Maximum frequency	313 MHz
	No. of 36k BlockRAM used	4 (6%)
	Dynamic Power	78.68 mW
ASIC	Area	10906 μmm^2
	Dynamic Power	107 μW

KB for storing BBs. For running multiple applications in the system, their corresponding BBs need to be stored on FPGA, which will require larger memory size. However, with the small area overhead incurred for the memory as shown in Table 5.8, our design can support multiple applications.

The exception handler landing pad address (EH) also needs to be stored along with the BB information. As said earlier, the number of EH addresses depends on the exceptions (*try-catch* and default exceptions) that have been handled in the program. As per our observation, the number of such EH addresses are not significantly large. For our modified benchmark program with one *try-catch* block, the number of EH addresses are 42. As the number of *try-catch* blocks is increased to 2 and 3, the number of EH addresses increases to 44 and 47 respectively. This shows that for a larger benchmark such as 403.gcc, the EH addresses account for <1% increase in the reference data. Thus, the EH addresses introduce a marginal increment in memory size.

As a proof of concept and measurement of CFC parameters, we implemented CFC module on the FPGA with BBT size of 16 entries, and RAS size of 16 entries. Table 5.8 summarizes the resource utilization of CFC implementation. It utilizes only small amount of resources of Virtex-5, i.e., <1% logic and 6% BRAM. The CFC has very small power consumption, a dynamic power of 78.68 mW, which makes it lightweight and suitable for embedded applications.

We also estimated the area and power for CFC on ASIC platform at 65 nm technology. As shown in Table 5.8, for BBT size of 1024 entries and RAS size of 16 entries, CFC has negligible area overhead of 0.02% (10906 μmm^2), while consuming a very low power of 107 μW. The processor die size is taken as 55 mm^2 for Intel processor [148], which has the same technology. The area will increase mainly due to increment in memory size. The number of basic blocks in complex programs will be large, which will linearly increment the memory size. For example, 403.gcc requires 209 KB (Table 5.7). As explained in Section 5.5, the main memory can be used as a secondary memory for storing the basic block information to reduce the memory area on FPGA. However, fetching data from the main memory will incur small overhead. The `call` instruction takes two FPGA cycles to validate: one clock cycle to verify the TA and one clock cycle to store the return address into the RAS. The `jmp` and `ret` instructions both take two FPGA cycles. The target address for `ret` is first searched in RAS and then in BBT for EH, whereas for `jmp`, it is first searched in RAS and then in BBT (as per the rules in Section 5.4.3).

Through simulation, we estimated the TAB buffer size and RAS stack size required for each benchmarks without causing processor performance overhead. TAB buffer size depends on the frequency of the CFIN in the program binary, i.e., the interval of the occurrence of CFIN during program execution, the processing speed difference between FPGA and processor. RAS size is determined by the consecutive number of `call` instructions in a program. CFC (running at 313 MHz) takes two FPGA cycles to perform verification. For processor running at 4 GHz, two FPGA cycles are equivalent to approximately 24 processor cycles. For 2 GHz and 1 GHz processors, two FPGA cycles are equal to 12 and 6 processor cycles respectively. Table 5.9 shows the maximum buffer size and RAS size requirement for processor running at 1, 2 and 4 GHz and CFC running on FPGA at 313 MHz.

The maximum buffer size requirement is larger for a 4 GHz processor. This is because the CFC takes 24 processor cycles to validate an entry. If the CFINs are more frequent and appear in less than 24 cycles interval, they get accumulated in TAB. The waiting time for the last entry to be processed by CFC gets increased.

Table 5.9: TAB and RAS sizes for zero performance overhead

Benchmarks	Ins. ($\times 10^6$)	Max. RAS	Max. TAB Processor (GHz) 4	2	1
djpeg	6.7	14	69	4	2
blowfish	65.29	12	30	4	0
susan.corners	1.12	11	52	4	2
401.bzip2	100	13	14	2	0
403.gcc	100	60	9765	12	2
462.libquantum	100	13	4779	2	0
483.xalancbmk	100	50	744	22	2
006.phylip	100	13	30	2	0
stream	100	11	30	2	2

As a result, the buffer size needs to be large enough to hold a substantial number of CFIN entries.

5.6.3 Performance Overhead

Our checking module CFC runs parallel to the processor. So ideally, there should be no performance overhead caused to the processor. However, once the buffer gets filled, the processor has to be stalled which results in performance overhead. Our results show that 403.gcc and 462.libquantum require comparatively a larger buffer size, assuming 24 processor cycles equal to 2 FPGA cycles (for 4 GHz CPU). We explored the performance overhead for these two benchmarks for different buffer sizes. The performance penalty in terms of percentage increase in execution cycles is shown in Fig. 5.11. Our experiments show a 0.6% performance penalty for buffer size of 1 KB. With the increasing buffer size, the performance overhead becomes negligible. However, the verification of the last entry gets delayed in the large queue. Whereas for 2 and 1 GHz processors, smaller buffer is required. For contemporary embedded system, where the processor to FPGA speed difference is comparatively less, our method is fast enough to detect attacks with very modest

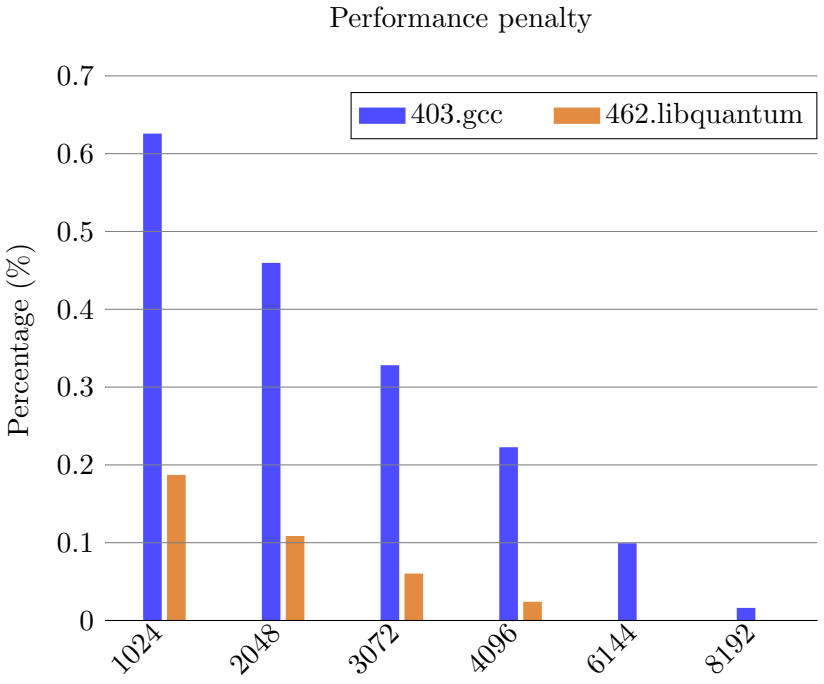


FIGURE 5.11: Estimation of Performance penalty on processor

execution overhead. The performance will be impacted only when the CFINs execute at a high frequency because once the buffer gets filled, the processor has to be stalled. Since we buffer the CFIN entries in the TAB buffer so that the processor can execute the instructions without stalling the pipeline, the SPEC benchmarks have <1% performance overhead (as shown in Fig. 5.11). Therefore, the performance will not be impacted significantly even for a large application.

5.6.4 Case Study

Our approach is generic and does not require source code. It can be applied on any embedded application with minor modification in hardware and using platform specification. An automated teller machine (ATM) machine is one example where our security approach can be applied. For our case study, we built a simulation environment for the ATM machine. It consists of μ-kernel running on top of Intel x86 processor. On top of μ-kernel, two separate partitions have been created for running user applications (e.g., front-end user interface, back-end application)

and system applications (e.g., networking, device drivers, health monitoring). Applications are spatially and temporally separated. We monitored the user interface program, which mainly handles the basic functionalities such as menu display and user input/output transactions. Similar to the real ATM machine, first, it authenticates the user and then performs the bank transactions by interacting with the back-end database application. The normal operations include the functionalities that are used by the general ATM users (e.g., user authentication, cash withdrawal/deposit, balance enquiry, ATM PIN modification) with valid input sets. False alarm will only occur if the application reaches a normal control path that has not been covered during profiling. However, most of the normal program paths can be profiled as the functionalities are well defined. We intentionally introduced a buffer overflow vulnerability in the code to exploit during runtime (as done in our shellcode attacks). At runtime, we exploited the system by performing ROP, JOP and stack smashing attacks by manipulating the user inputs (using similar techniques as our shellcode programs). Our approach easily detects these attacks and raises the alarm. Since these attacks follow the general notion of the real world attacks, all similar code reuse and stack smashing attacks will be detected by our approach.

5.7 Summary

In this chapter, first we presented a fine-grained CFI policy at basic block level, named BB-CFI, which aims to defend against runtime attacks – stack smashing, ROP and JOP. Our method involves an offline profiling of the program and runtime verification of the control flow. BB-CFI validates the target address of the control flow instructions including `call`, `ret` and `jmp`, which are responsible for diverting the program execution, using the reference data obtained during profiling. Second, we presented the architecture of control flow checker (CFC) to enforce BB-CFI policy at runtime.

Compared to other techniques, BB-CFI effectively enforces fine-grained CFI at the basic block level. Besides, it does not require ISA or code (source or binary) modification. The CFC design is prototyped in FPGA. The results show <1% performance overhead for a 4 GHz CPU. Since the overheads (performance, area) and power consumption are low, we conclude that our design can fit well into embedded systems domain. In future, we plan to implement our design on the multiprocessor systems. We will employ an advanced program analysis technique to increase the coverage of the control flow graph.

6

Defense against ROP Exploits using Hardware Performance Counters

6.1 Introduction

Return-oriented programming (ROP) is the most widely-used offensive technique to exploit software vulnerabilities. According to the recent Microsoft's software vulnerability exploitation trends report [149], 80% of all vulnerabilities are exploited using code-reuse or ROP attacks. Further, zero-day ROP exploits are emerging against nearly all commercial off-the-shelf software products such as Adobe Flash Player [150] and Internet Explorer [151], and popular open-source software such as Firefox [152]. A ROP attack exploits the presence of *gadgets* (i.e., small instruction sequences ending in a return instruction) in the target

program, where multiple gadgets are all chained together to build complex yet meaningful attacks.

To deliver the ROP payload, spraying attack, first introduced in 2004 [153], consistently remains one of the most commonly-used mechanisms [149]. It can place attacker-controlled code or data at a desired memory location. Since the inception, many high-profile attacks including the exploits in Internet Explorer [154] and Adobe Reader [155] leverage spraying attack techniques to accomplish their attacks.

ROP is so powerful that it can easily evade recent mitigation techniques such as data execution prevention (DEP) [156]. DEP or W⊕X is a hardware protection mechanism that implements non-executable memory in order to protect the execution of injected code. However, ROP can still thwart DEP, as it reuses existing code sequences instead of the injected code. ASLR [111] is another OS mitigation technique that randomizes the stack and base addresses of most of the libraries and executables to defend against memory corruption attacks. However, ROP attack can still be performed by using the statically known locations (e.g., using memory disclosure attack [80]). While so many defense techniques have been proposed, there has also been a continual evolution in sophistication of ROP exploits to defeat the state-of-the-art defense mechanisms [70, 82, 157].

The existing ROP exploit defense techniques suffer from several key problems. As a result, most techniques can be easily defeated by attackers through carefully crafting code-reuse attack payloads, or require high performance overhead. For example, most ROP mitigations rely on API hooking [84, 135] or instrumentation [83, 158, 159] to detect ROP attacks. However, hooking-based techniques can be easily bypassed by hook hopping [160], while instrumentation-based techniques have high performance overhead. Besides, some of the defensive mechanisms leverage predefined policies to detect ROP attacks [135]. However, these policies are inflexible and weak, and hence can be bypassed by constructing ROP gadgets that adhere to such policies. Also, the runtime policy checking usually incurs high performance overhead, making them impractical for real-world deployment.

Besides, the state-of-the-art spraying attack detection mechanisms [161, 162] can only handle attacks that use system memory allocators, while failing to detect attacks that use custom memory allocators [163]. Unfortunately, custom memory allocators are becoming popular among software applications in the recent years, which alleviates the applicability of these techniques.

Differently, several other ROP mitigations [135, 164–167] leverage low-level hardware events in the CPU (e.g., cache misses and branch mispredictions) via hardware performance counters (HPCs) to detect code-reuse attacks. Compared to software behaviors, these hardware events are much more difficult for attackers to control directly in evasion attacks. For example, it is easier for attackers to change system calls than to manipulate branch mispredictions in a precise way while performing exploits. However, these techniques suffer from two problems: (1) high performance overhead resulting from the high-frequent interrupts to obtain HPC values and the heavy-weight machine learning techniques to process these HPC values; and (2) limited availability of programmable performance counters in the CPU [167] that only allows limited events to be monitored at runtime.

To address these problems, in this work, we propose a novel defense framework, called *ROPSentry*, to detect ROP attacks at runtime using HPCs with low performance overhead. The key observation underlying ROPSentry is that a ROP attack triggers hardware events in a significantly different way than a normal program generated by a compiler. For example, ROP exploits chain several gadgets ending in return instructions and cause an unexpected program control flow, leading to high branch misprediction. Similarly, a spraying attack usually performs similar memory allocation operations, leading to similar performance counter values for load microinstructions. Therefore, a ROP attack can be distinguished from a normal program at runtime by capturing the behavioral patterns of spraying attacks and/or ROP exploits from monitored hardware events.

However, ROP gadgets are small in nature, i.e., around 3-5 instructions per gadget. Hence, to capture any malicious pattern, we need to fetch HPC values in a high frequency, which increases the performance overhead during runtime. To

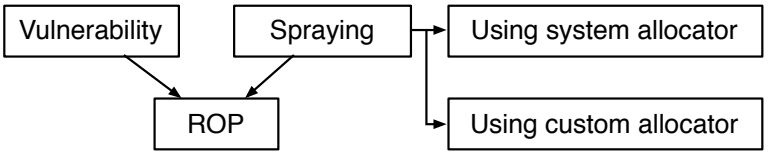


FIGURE 6.1: Variants of ROP Attacks

tackle this problem, we develop a *ROP-only defense approach* into our ROPSentry framework, which detects ROP attacks by capturing the patterns of ROP exploit precisely through sampling HPCs at mispredicted return events instead of at every microinstruction, based on the observation that ROP exploits cause several return mispredictions. To further reduce its performance overhead, we introduce a *self-adaptive defense approach*, which leverages the ROP payload delivery mechanism spraying attack (which often involves millions of instructions), to dynamically switch between two sampling rates at runtime such that the performance overhead is negligible. In particular, we initially employ a low sampling rate to detect the spraying attack. Once a potential spraying attack is observed, we switch to a high sampling rate to determine whether a ROP attack has occurred. If yes, we terminate the program; otherwise, we switch back to the low sampling rate to continue the detection of spraying attack. Thus, our framework can detect both non-spraying-based ROP attacks and spraying-based (using either system or custom allocators) ROP attacks. This work provides a complete solution to handle the entire spectrum of ROP attacks, as shown in Fig. 6.1 (refer to Section 6.2.1 for a detailed discussion).

We evaluated the detection accuracy of our framework on 11 real-world ROP exploits and 50 synthetically generated ROP exploits. The results indicated that our framework can detect all the 61 exploits. We also evaluated the performance overhead and the false positive rate of our framework using 1000 benign websites, which demonstrated that our framework can keep the performance overhead at 11% and 1% when using the ROP-only and self-adaptive defense approach respectively, and have zero false positive. Moreover, we compared our framework with several state-of-the-art techniques, including EMET [168], the approach in [167], and Graffiti [161]. The results demonstrated that our framework had much lower

performance overhead than theirs without losing the detection accuracy.

The main contributions of this work are as follows.

- We use hardware events to capture behavioral patterns of ROP exploits, based on which we propose a ROP-only defense approach to detect both non-spraying-based and spraying-based ROP attacks at runtime.
- We propose a self-adaptive defense approach to further reduce the performance overhead of detecting spraying-based ROP attacks, i.e., dynamically switching between low and high sampling rate to detect spraying attacks and ROP exploits.
- We empirically demonstrate that, ROPSentry is effective in detecting ROP attacks at runtime with low performance overhead and low false positive; and it significantly outperforms the state-of-the-art techniques in terms of performance overhead without sacrificing the detection accuracy.

The rest of the chapter is organized as follows: Section 6.2 gives the preliminaries and challenges in existing ROP mitigations. Section 6.3 reviews the related work. Section 6.4 introduces the proposed approaches. Section 6.5 explains our implementation details. Section 6.6 evaluates the proposed approaches, and Section 6.7 makes some discussion. Section 6.8 draws the conclusions.

6.2 Preliminaries and Challenges

In this section, we first briefly introduce the preliminaries on ROP attacks and hardware performance counters (HPCs), and then summarize the challenges in existing ROP mitigations.

6.2.1 ROP Attacks and the Variants

Fig. 6.2 presents the typical program memory layout of a ROP attack. ROP uses small instruction sequences ending in a return instruction, called gadgets. The

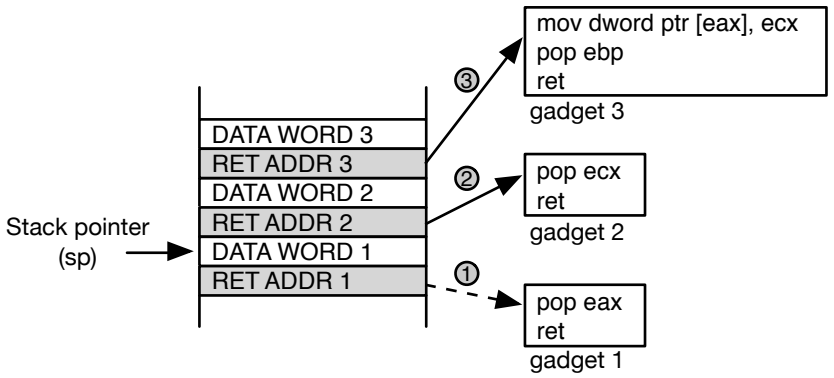


FIGURE 6.2: The Memory Layout of a ROP Exploit

ROP payload consists of return addresses that target the gadgets already present in the library or victim code. The stack pivot gadget will switch the original stack to the attacker-controlled data in the memory via triggering a vulnerability (e.g., Use After Free) in the application to pass the control to the attacker-controlled ROP chain. The execution of the return instruction in the gadget moves the stack pointer to the next return address. The stack pointer advances the control from one gadget to another. Each gadget contributes certain computations to perform the complete attack.

Fig. 6.1 shows the variants of ROP attacks, which can be (1) non-spraying-based ROP attacks and (2) spraying-based ROP attacks. Non-spraying-based ROP attacks are possible via exploiting vulnerabilities such as heap overflow. Attackers first put the ROP payload in the heap, and then exploit the heap vulnerability to control the stack pointer and program counter. Differently, attackers can leverage on spraying attacks to spray the ROP payload in the heap, and employ some offensive techniques such as stack pivoting to execute the ROP payload. They can make use of either system allocators (e.g., malloc) or custom allocators [169] (e.g., Jemalloc and Oleaut32) to allocate memory for achieving the spraying.

6.2.2 Hardware Performance Counters

Performance monitoring unit (PMU) was initially introduced in Pentium processor with a set of model-specific performance-monitoring counters [170]. It allows the

Table 6.1: Monitored Hardware Performance Events

Architectural Event	Description
1. Ins	instructions retired
2. Clk	unhalted core cycles
3. Br	branch instructions
4. Arith	arithmetic instructions
5. Call	all call instructions
6. Call_D	direct near call instructions
7. Call_ID	indirect near call instructions
8. Ret	near return instructions
Non-architectural Event	**Description**
9. Uops	micro-instructions retired
10. Load_Uops	load uops retired
11. Store_Uops	store uops retired
12. Br_Miss	mispredicted branch instructions
13. Ret_Miss	mispredicted return instructions
14. Call_D_Miss	mispredicted direct call instructions
15. Call_ID_Miss	mispredicted indirect call instructions
16. Br_Far	far branches retired
17. ITLB_Miss	misses in all ITLB levels
18. DTLB_Store_Miss	store uops with DTLB miss
19. DTLB_Load_Miss	load uops with DTLB miss
20. ICache_Miss	instruction cache misses
21. LLC-Ref	longest latency cache reference
22. LLC_Miss	longest latency cache miss
23. LLC_Load_Miss	load uops with LLC miss
24. L2_Code_RD	L2 instruction cache access
25. L2_Code_Miss	instructions that missed L2 cache
26. Load_Uops_L1_Hit	load uops with L1 data hits

selection of performance parameters that can be monitored and measured. This hardware-level CPU feature provides a mechanism for software to monitor and count performance events of interest. The initial aim was to use the information obtained from these counters to tune the performance of systems and compilers.

PMU supports two kinds of performance monitoring capability: architectural and

non-architectural (i.e., micro-architectural). Architectural performance monitoring supports the counting and sampling of a small set of performance events such as arithmetic and branch instructions. These events are consistent across different processor implementations. Differently, non-architectural performance monitoring monitors a large set of performance events that are specific to the micro-architecture, such as cache, branch prediction and translation lookaside buffer (TLB) (i.e., micro-architectural events). These events might differ from one processor to another and might change with processor enhancements. The complete list of architectural and micro-architectural events is available at Intel's manual [170]. Here we only list the performance events that are used in this work in Table 6.1.

6.2.3 Challenges in Practical ROP Defense

Numerous exploit defense techniques have been proposed to mitigate ROP attacks. However, several challenges still remain.

C1. Hooking-based techniques can be bypassed by hook hopping. Some ROP mitigations [84, 135] rely on API hooking to detect ROP attacks, where the hook redirects the function in order to verify certain properties. API hooking techniques are also used by commercial security software such as EMET [168] and antivirus tools. However, such techniques are not robust as they can be easily bypassed by hook hopping techniques.

Fig. 6.3 shows a code snippet to illustrate how a few lines of assembly code can be used to defeat the inline function hooking mechanism. The shellcode contains its own pre-hook preamble. After its execution, it transfers the execution to skip the function hook *Hook_VirtualProtect*. Similarly, ROP exploit can leverage memory information disclosure to detect API hook and utilize specific ROP gadgets to bypass hooks during the exploit phase.

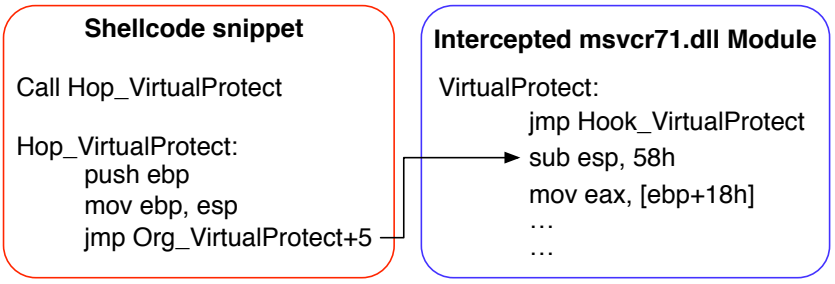


FIGURE 6.3: Hopping the *Hook_VirtualProtect* Hook

C2. ROP detection policies are inflexible and weak. Some ROP mitigations [135] use predefined policies to detect ROP attacks. In general, such policies are inflexible and weak, and thus can be easily bypassed through constructing the ROP gadgets (or payloads) that adhere to these policies.

For example, the *call-preceded return address* policy specifies that return instructions need to target a valid call-site (a call-preceded instruction), which may include non-intended call instructions. However, this policy can be defeated, as the call-preceded ROP gadgets are turing-complete and therefore, real-world ROP exploits can still be launched by conforming to this policy [82]. Similarly, another commonly-used policy is the *chain of short sequences*, where a ROP attack is reported after N sequences each consisting of less than S instructions. Unfortunately, this policy can be bypassed by introducing new gadget types such as the long no-operation gadget, where the long sequence does not break the semantics of the ROP chain [82].

C3. The performance overhead of the existing detection approaches is high. Both instrumentation-based [83, 158, 159] and policy-based [135] mitigations have high performance overhead because of the inefficient instrumentations and runtime policy checking. Similarly, most HPC-based techniques [135, 164–167] have high performance overhead during online detection due to two main reasons. First, they use a high sampling rate, which significantly creates high-frequent interrupts to fetch HPC values and thus hinders its practical applications. Second, almost all the HPC-based ROP detection approaches use some machine learning techniques, which cause significant runtime overhead as the classifier has to validate the trace

expensively at each interrupt. Hence, most of the techniques do not implement it at runtime, but use it during offline analysis.

It is also worth mentioning that the machine learning-based techniques need a large amount of data for training and model building. However, the number of ROP exploits in real world is significantly small, as compared to malware samples. Also, the exploits are very distinct from each other, which limits the practicality of these machine learning-based approaches. In addition, a small manipulation in the exploit technique can significantly change the features that are used by machine learning approach, thus, making such defense approach ineffective [171].

C4. The resource is limited. All the HPC-based defense techniques are limited by the available resource w.r.t. the programmable performance counters in the CPU. This is because, current processors generally have 2-4 programmable performance counters per thread [170]. In that sense, all previous approaches can only monitor four hardware events in addition to the fixed events (by using fixed performance counters) to model the attack patterns.

C5. Custom allocator-based spraying attacks are difficult to detect. In existing spraying attack detection techniques [161, 162], it is not possible to detect spraying attacks that make use of custom allocators, which generally allocate a big chunk of memory at the beginning of the process and then use their own allocation functions to perform memory operations. Unfortunately, custom allocators are quite common in real-world applications, which alleviate the applicability of those detection techniques.

6.3 Related Work

In this section, we focus our discussion on the exploit defense techniques that perform runtime defenses, and those techniques that use hardware features for ROP detection.

6.3.1 Instrumentation- and Hook-Based Runtime Monitoring

Since the inception of ROP attacks by Shacham [65], several defense techniques have been proposed. As a response to these defense techniques, ROP attacks have evolved in their sophistication, in order to bypass them [70, 82, 157]. Initial runtime ROP solutions focus on using instrumentation or hooking techniques. For example, they monitor program at the instruction level using dynamic binary instrumentation frameworks, such as PIN [172]. Towards this, Davi et al. [158] proposed the runtime integrity monitoring technique to detect ROP exploits by checking for gadgets with small number of instructions. DROP [81] builds its defense mechanism against traditional ROP exploits based on the heuristics that these exploits contain long consecutive sequence of return instructions, and the gadgets are small in length. As discussed by *C2* in Section 6.2.3, long gadgets can be used to defeat these techniques. ROPDefender [83] instruments call and return instructions and maintains a shadow stack to verify that the return address is present in the shadow stack. ROPGuard [84] uses heuristics to detect ROP exploits, such as instruction preceding the return address must be a call instruction. Its main drawback is that it performs the validation only when a critical Windows function is called. This makes it vulnerable to ROP attacks that can jump over these checks (e.g., using hook hopping as discussed in *C1* in Section 6.2.3) and those ROP attacks that do not use any critical Windows function [82].

6.3.2 Hardware-Based Program Monitoring

Recently, several hardware-based solutions have been proposed in the literature to overcome the limitations in program instrumentation and hooking based ROP mitigation techniques. One of the pioneers in this line of work is kBouncer, proposed by Pappas et al. [135], that leverages on the Last Branch Record (LBR) facilities available in modern CPUs to record the branch targets of the last 4-16

branch instructions taken by the target program. At each system API invocation, kBouncer verifies the integrity of the running program by evaluating the proposed CFI policies against the LBR stack. Later, ROPecker [164] extended the idea proposed in kBouncer by offline analysis and emulation in an attempt to predict ROP attacks. However, as discussed by *C2* in Section 6.2.3, weak CFI policies can be easily defeated by specially crafted ROP gadgets (e.g., call-preceded gadgets) and further, API hooking is not a robust technique as it can be evaded by hook hopping (see *C1* in Section 6.2.3).

Yuan et al. [165] studied on how Branch Trace Store (BTS), a debugging mechanism that allows to record all branch instructions, along with several other HPC features such as ITLB misses and branch misses can be used to detect code injection attacks. Unfortunately, this approach incurs heavy performance overhead due to the significant number of memory accesses. Similarly, Wicherski [166] used single branch prediction event as an indicator to detect kernel-level ROP attacks. However, it is shown that all of these techniques are vulnerable to several evasive techniques such as those in [70] [82] [157]. Their usage on a single type of feature (i.e., HPC event) is the root cause for their incapabilities. Demme et al. [173] and Malone et al. [174] used the HPC data profiles to extract malware signatures and detect program integrity violations respectively. Both techniques are orthogonal to our approach as they focused on intrusion detection instead of exploit detection. Tang et al. [167] proposed a system to detect drive-by attacks in Internet Explorer by leveraging HPC data combined with machine learning. However, they focused more on the post-ROP behavior (i.e., malicious behavior) of the program, where they reported a high malicious behavior detection rate but a low ROP attack detection rate. Graffiti [161] is the most recently proposed approach, which implements a hypervisor-based memory analysis technique for detecting spraying attacks. As discussed in Section 6.6.5, its main limitation is that they cannot handle custom allocator-based spraying attacks and hence it cannot protect against the modern spraying-based attacks. Besides, their approach incurs much higher overhead (23%) as compared to our approach (1%).

6.3.3 CFI-Based Defenses

Control flow integrity (CFI) was first proposed by Abadi et al. [72] to prevent from software runtime attacks by enforcing program execution to follow its predefined control flow graph. Although the technique is quite promising, the main difficulty is to obtain a precise control flow graph of the program. Because of this, many techniques implement weak CFI policy, for example, allowing returns to be call-preceded, indirect calls or jumps to target function entry [74, 75, 126, 175]. These CFI-based techniques have been shown ineffective, since the ROP exploits can still be performed by following the policies in [70, 71].

6.4 Methodology

To address the challenges in Section 6.2.3, we propose our ROPSentry framework, which employs a *ROP-only defense approach* (Section 6.4.1) and a *self-adaptive defense approach* (Section 6.4.2). ROPSentry is built on the observation that the execution of a ROP attack triggers hardware events in the CPU in a significantly different way than a normal program generated by a compiler. Such hardware events are side channels that can be easily monitored through the HPCs present in every modern CPU. Therefore, a ROP attack can be distinguishable from a normal program at runtime by capturing the patterns of ROP attack from monitored hardware events. In that sense, we address the challenges *C1*, *C2* and *C5* by leveraging hardware events that are much more difficult for attackers to be controlled directly in evasion attacks than software behaviors.

Scope. Before elaborating ROPSentry, we first introduce the scope of this work. We focus on both non-spraying-based ROP attacks, and spraying-based ROP attacks that use custom or system allocators, and thus provide a complete solution against the entire spectrum of ROP attacks.

6.4.1 ROP-Only Defense Approach

In this approach, ROPSentry fetches the HPC values with a sampling rate (i.e., s_h) to detect the patterns of ROP exploits (see Section 6.4.1.1). If we observe that a ROP exploit pattern is occurred, we detect a ROP attack and terminate the program.

However, the sampling frequency to fetch HPC values heavily influences the practicality of the approach, i.e., a high sampling frequency improves detection accuracy at the price of performance overhead, while a low sampling frequency reduces performance overhead at the cost of detection accuracy. Based on the fact that return misses are very common in ROP attacks but not very common in normal programs because of the well-designed branch prediction unit [176], we propose a novel return miss-based sampling technique to sample the hardware events, i.e., fetching HPC values at every several return misses.

Different from previous approaches [167], which mainly use a microinstruction-based sampling technique, our approach can clearly differentiate between ROP chain execution and normal program execution and has lower performance overhead. For example, 21,918 microinstructions, on average, are executed for every 10 return misses during the whole ROP attack (for the ROP exploits we used in the evaluation), while only 30 microinstructions are executed for every 10 return misses during the specific ROP chain. Therefore, even the ROP chains that have only 30 microinstructions can be detected by the return miss-based sampling. However, if the events are monitored at every 100 microinstructions, the performance overhead is increased by about 200 times. In this way, we address the challenge *C3*.

6.4.1.1 ROP Exploit Pattern

A ROP exploit often executes multiple gadgets to perform arbitrary computations. Each gadget is a small instruction sequence that ends mostly in a branch instruction (e.g., call, return, and jump). Most of the current ROP exploits use gadgets

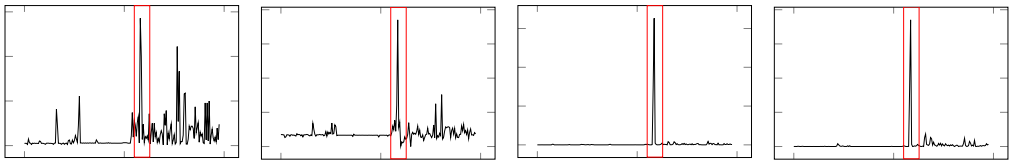


FIGURE 6.4: An Example of ROP Exploit Behavior

ending in a return instruction, which often cause an unexpected control flow of the program. This results in high branch mispredictions during the exploit-code execution. However, the branch prediction unit in modern processors is well designed to achieve high prediction accuracy (generally $> 95\%$). Based on such understanding, we select 26 relevant events listed in Table 6.1 in order to model the behavior of ROP exploits. We run five normal programs as well as five real-world ROP exploits from the Metasploit framework [177] to determine the best set of hardware events that can be used to model the behavior of ROP exploits.

It can be commonly observed from the five ROP exploits that there is a *simultaneous* spike, as highlighted by a red rectangle in the first five rows in Fig. 6.8, during the exploit-code execution for the values of performance counter *Ret_Miss*, *Ret*, *ITLB_Miss* and *LLC_Miss* after the normalization by *Ret* or *Ins*. It is only observed in ROP attacks but not in normal programs, and thus can be used to clearly differentiate the ROP exploit from the rest of the code execution. For the fifth row in Fig. 6.4, there is a second spike for the metric *Ret_Miss/Ret*, which does not occur simultaneously in the other three metrics. It indicates that *a combination of metrics can effectively reduce false positive.*

Specifically, ***Ret_Miss/Ret*** represents the percent of mispredicted return instructions in each sampling interval. The branch prediction unit uses the call stack at runtime to predict the return address for each function return. However, a ROP exploit manipulates return instructions to execute arbitrary computations and causes an abnormal program control flow. Since a ROP exploit occurs over a small length of instructions, the return miss rate in this small length becomes substantially high. As observed from the five ROP exploits, the return miss rate is in the range of 0.9 and 1.3. This means the return miss rate is close to 100% during a ROP exploit,

which is a strong evidence to justify that a ROP exploit might be performed. Note that the return miss rate may exceed above 1.0 because *Ret_Miss* is speculative but *Ret* is the number of actually executed return instructions, and not all the return instructions that are speculated get executed in the execution stage of CPU pipeline. Hence, we empirically set the threshold value of return miss rate to 0.9 to detect ROP exploits, i.e., the policy 1: $Ret\_Miss/Ret \geq 0.9$.

Ret/Ins measures the percentage of return instructions in each sampling interval. As ROP exploits use return instructions to orchestrate the ROP control flow (i.e., to redirect the control flow from one gadget to another), the rate of return instructions becomes substantially high during the ROP chain. For the five ROP exploits, this ratio is between 0.2 and 0.4. Thus, we empirically set its threshold value to 0.2, i.e., the policy 2: $Ret/Ins \geq 0.2$.

ITLB_Miss/Ins measures the percent of misses in all instruction translation lookaside buffer (ITLB) levels over the total number of instructions in each sampling interval. ITLB is a cache that memory management hardware uses to improve the virtual address translation speed. It is well designed in modern processors such that the ITLB miss rate is very low, i.e., around 0.01–1% [178]. However, the gadgets in a ROP exploit are already present in the original program and spread over different program segments. As a result, the executed ROP chain code is spread over too many pages and causes many ITLB misses, and thus the ITLB miss rate is substantially high. As observed from the five ROP exploits, this ratio is between 0.8 and 1.6 during the ROP chain. Therefore, we set its threshold value to 0.8 to detect ROP exploits, i.e., the policy 3: $ITLB\_Miss/Ins \geq 0.8$.

LLC_Miss/Ins represents the percent of misses in the last level cache (LLC) over the total number of instructions in each sampling interval. LLC is the last cache before RAM in the memory hierarchy. An abnormal control flow during ROP exploit leads to branch mispredictions and ITLB misses. While handling an ITLB miss, initial cache levels and LLC are accessed, first for page table entry and later for instructions. LLC misses occur because of the unavailability of data in LLC, eventually leading to fetch data from main memory or disk. Thus, the

underlying reason for the high LLC miss rate in ROP exploits is their unexpected program control flow. In the case of the five ROP exploits, the LLC miss rate is in the range of 2.0 and 3.0. This ratio exceeds above 1.0 because LLC miss occurs not only for instructions but also for data. We conservatively set the threshold value of LLC miss rate to 2.0, above which a ROP exploit is detected, i.e., the policy 4: $LLC\_Miss/Ins \geq 2.0$.

Based on these observations from the five ROP exploits, we determine the best set of hardware events to model the behavior of ROP exploits as *Ret_Miss*, *Ret*, *ITLB_Miss*, *LLC_Miss* and *Ins*. Based on these monitored events, we specify the pattern of ROP exploits in Pattern 1.

Pattern 1. All the four policies about *Ret_Miss/Ret*, *Ret/Ins*, *ITLB_Miss/Ins* and *LLC_Miss/Ins* are *simultaneously* satisfied in a given sampling interval at runtime.

The ROP-only defense approach can achieve a high detection accuracy of ROP exploits but at the cost of high but still acceptable performance overhead. Note that the performance overhead is lower than the previous approaches [161, 167] (see Section 6.6). To further reduce the overhead, we propose a self-adaptive defense approach below without sacrificing the detection accuracy.

6.4.2 Self-Adaptive Defense Approach

The self-adaptive approach aims to reduce the performance overhead of our ROP-only approach, by first sampling at a low frequency to detect spraying attack that is mostly used by ROP exploits for payload delivery. The workflow is shown in Fig. 6.5 as a state transition diagram. In this approach, the ROP attack, when executed, can be in any of the following states: (1) normal state, (2) ROP state, and (3) termination state. In the normal state, we fetch the HPC values with a low sampling rate (i.e., s_l) to detect the pattern of spraying attack (see Section 6.4.2.1). If we observe a spraying pattern, we suspect that a spraying attack occurs and move to the ROP state. In the ROP state, we fetch the HPC values with the high sampling rate (i.e., s_h) to detect the potential pattern for any ROP

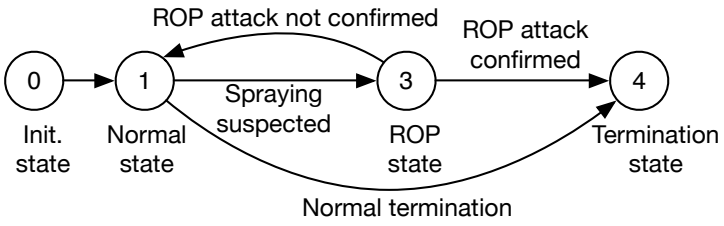


FIGURE 6.5: Workflow of Our Self-Adaptive Defense Approach

exploit (see Section 6.4.1.1). If we observe that a ROP exploit pattern is occurred, we detect the ROP attack and terminate the program; otherwise, we move back to the normal state and switch back to the low sampling rate. Here the two sampling rates are empirically determined as will be discussed in the evaluation.

Based on the facts that, spraying attack is a common ROP payload delivery mechanism, and a spraying usually involves millions of instructions while ROP gadgets contain around 3-5 instructions, we design this self-adaptive sampling technique that respectively employs low and high sampling rates to detect the patterns of spraying attack and ROP exploit and dynamically switches between them. In this way, we address the challenge *C3*, i.e., keeping performance overhead at an even lower level.

Meanwhile, our self-adaptive sampling technique also dynamically configures the four programmable performance counters in the CPU to monitor two different sets of hardware events, i.e., one for detecting spraying attacks and the other for detecting ROP exploits. In this way, we address the challenge *C4*, i.e., breaking the limit of four programmable performance counters.

6.4.2.1 Spraying Attack Pattern

A spraying attack typically fills the heap with copies of NOP sled and shellcode that are large enough for the shellcode to land at a predetermined address and to execute arbitrary code after the exploitation of a vulnerability. Therefore, a spraying attack usually performs similar memory operations and executes similar code for a number of times. Based on this understanding, we select 26 relevant performance counters (including those for ROP exploits), as listed in Table 6.1;

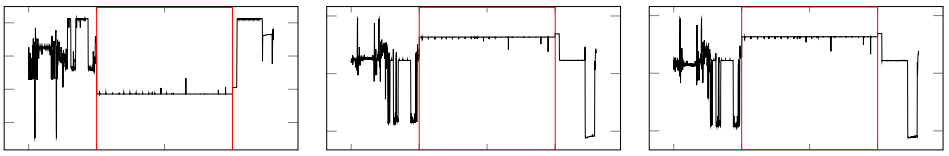


FIGURE 6.6: An Example of Spraying Attack Behavior

and run five normal programs and their corresponding ROP attacks using five real-world ROP exploits from the Metasploit framework [177] to determine the best set of hardware events that can be used to model the behavior of spraying attacks.

The commonly observed behavior across the five spraying attacks used in the five ROP exploits is that, the values of performance counter *Br*, *Load_Uops* and *Load_Uops_L1_Hit* after the normalization by *Ins* or *Uops*, all keep nearly constant during the spraying attacks, as highlighted by the red rectangles in Fig 6.6. Further, this is only observed in spraying but not in normal programs, and hence can be used to clearly differentiate the spraying attack from the rest of the program execution.

In particular, ***Br/Ins*** represents the percentage of branch instructions in each sampling interval. Its value keeps consistent during the spraying attack process due to the execution of similar code for a number of times. ***Load_Uops/Uops*** and ***Load_Uops_L1_Hit/Uops*** respectively represent the percentages of load microinstructions and load microinstructions with L1 cache hit in each sampling interval. Their consistent values during the spraying attack are explained by the similar memory allocation operations to fill the heap.

Limited by the four programmable performance counters in the CPU, we determine the best set of hardware events to model the behavior of spraying attacks as *Br*, *Load_Uops*, *Load_Uops_L1_Hit*, *Uops* and *Ins*, where *Ins* can be monitored by the fixed performance counter in the CPU. Based on these monitored events, we specify the spraying pattern in Pattern 2.

Pattern 2. *Br/Ins*, *Load_Uops/Uops* and *Load_Uops_L1- _Hit/Uops* remain consistent *simultaneously* during a given time window (i.e., w_{hs}) at runtime.

w_{hs} is empirically determined as discussed in the evaluation. Once in the normal state, we use a sliding window algorithm to check the consistency during w_{hs}. In a given window, we observe if the ratio of the previous value to the present value of counter remains around 1, at a given sampling interval. To remove the noise, we ignore some anomalous cases, which may not be consistent in a given window. If the consistency lasts for w_{hs}, we suspect that a spraying attack occurs and move to the ROP state, where we use Pattern 3 below to specify the ROP pattern that is an extension to Pattern 1. If there is an inconsistency of the counter values for the monitored events during a given window, we move the position of the window to the next sampling interval. The only difference between Pattern 1 and 3 is that the policies will be checked for a time window w_{rop} instead of the whole running time of the ROP attack, since we need to switch back to the normal state.

Pattern 3. All the four policies about *Ret_Miss/Ret*, *Ret/Ins*, *ITLB_Miss/Ins* and *LLC_Miss/Ins* are *simultaneously* satisfied in a given time window (i.e., w_{rop}) at runtime.

w_{rop} is empirically determined as discussed in the evaluation. Once in the ROP state, at each interrupt, we check the satisfaction of the four policies for ROP detection. If satisfied, we confirm that a ROP exploit occurs; if not satisfied in w_{rop}, we move back to the normal state.

Here we clarify that our approach can only suspect but cannot confirm spraying attacks because we do not observe the content in the heap memory. Inspecting memory content at runtime will significantly impact the performance. Instead, we employ a conservative approach, i.e., every occurrence of the pattern is suspected as a spraying attack, but we do not flag it as an attack. We rather detect ROP exploit pattern in order to confirm the ROP attack. This approach is meaningful as a spraying attack is a payload delivery mechanism used by attackers to assist in executing the ROP chain.

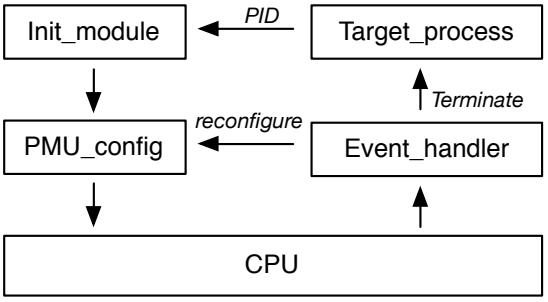


FIGURE 6.7: The Implementation Architecture of Our Approach

6.5 Implementation

We implement a prototype of the proposed approach on x86, a 32-bit version of Windows 7 Professional SP1. Fig. 6.7 presents the architecture of our implementation, which mainly consists of *Init_module*, *PMU_config* and *Event_handler*.

Init_module operates in the userland space. It obtains the process ID (PID) of a target process that needs to be monitored, and sends the PID to *PMU_config*. These two components communicate through control messages over a pseudo-device that is exported by *PMU_config*.

PMU_config operates in the kernel space. It is responsible for two main tasks, the initialization of PMU and the configuration of performance monitoring interrupt (PMI) through configuring the model specific registers (MSRs) by *rdmsr* and *wrmsr* instructions. The first task configures *IA32_FIXED_CTR_CTRL*, *IA32_PERFEVTSELx* and *IA32_PERF_GLOBAL_CTRL* to control what events to be monitored based on the state of the target process (i.e., the normal state or ROP state), and also configures *IA32_FIXED_CTRx* and *IA32_PMCx* to count the exact number of the monitored events. Each performance monitoring counter *IA32_PMCx* is paired with a performance monitoring select register *IA32_PERFEVTSELx*. The second task of *PMU_config* configures *IA32_PERF_GLOBAL_OVF_CTRL*, *IA32_PMC0*, *IA32_PERFEVTSEL0* and *IA32_PERF_GLOBAL_STATUS* to generate a PMI for every N events, where N is determined by the sampling rate s_l and s_h.

There are two implementation challenges for *PMU_config*: 1) how to handle each PMI and 2) how to separate the PMU for the target process from the other processes in the whole system. For the first challenge, Windows OS does not provide any documentation to register a call back function to handle the interrupt. The potential possibilities are to use interrupt descriptor table (IDT) hooking, API hooking, or callback hooking [160]. In this work, we use the IDT hooking technique to hook each PMI for collecting and processing the PMU data, i.e., to redirect the IDT hook vector *0xFE* handler to our custom code. The IDT hook needs to preserve the original interrupt context after PMU data collection and processing. Finally, we redirect the interrupt to the default OS handler. IDT hooking is not allowed in 64-bit Windows OS due to the implementation of a mitigation technique called PatchGuard, which restricts kernel patching. Alternatively, callback function for PMU can be registered using *HalpSetSystemInformation* API [179].

To address the second challenge, we first record the CR3 register value that is specific to the target process in the initialization step; and then at each PMI, we read the CR3 register value and compare it against the original CR3 register value recorded in initialization. In this way, we can collect the PMU data that is specific to the target process. However, the MSR registers that store the monitored information and configuration parameters are part of the running process context, and are preserved during context switches. Since we are not able to monitor the context switch, we may have some contaminated PMU data that may be related to other process. Note that this is not a design limitation but an implementation issue. This issue is caused by the fact that Windows OS does not preserve PMU data for a specific process and also it does not allow the monitoring of context switches. However, in Linux, PMU data is preserved for a specific process during context switches. This issue can be avoided in the future by preserving the counter data during context switches by making use of asynchronous procedure calls [179].

Event_handler analyzes the collected PMU data at each PMI to detect spraying attack or ROP exploit pattern. Specifically, if the target process is in the normal state (with a low sampling rate), it uses the sliding window algorithm to detect spraying attack pattern. Once a spraying attack is observed, it triggers

PMU_config to reconfigure performance counters with different set of events in order to switch to the high sampling rate and detect ROP exploit pattern. If a ROP exploit is not observed, *Event_handler* triggers *PMU_config* to switch back to spraying attack detection. In case a ROP exploit is observed, the target process is terminated and the user is alarmed.

To validate that our HPC patterns match the behavior of ROP exploit and spraying attack we also log windows APIs in addition to the HPC data during offline analysis. We only use it for our off-analysis on ROP exploit and spraying attack to confirm their presence. However, in our final approach we rely on HPC data only to detect ROP exploit and spraying attack. Specifically, we log several kernel32.dll APIs, including *HeapCreate*, *HeapDestroy*, *HeapAlloc*, *HeapFree*, *VirtualProtect*, *VirtualAlloc*, *VirtualProtectEx*, *CreateProcessA*, *CreateProcessW*, *WinExec*, *ReadProcessMemory*, *WriteProcessMemory*, *GetProcAddress*, *CreateThread*, and *CreateRemoteThread*.

6.6 Evaluation

In this section, we evaluate the detection accuracy and performance overhead of the proposed ROPSentry framework. All the experiments were conducted on a virtualized environment with the following specifications: Intel Xeon Processor, 8GB RAM, and 32-bit Windows 7 Professional SP1. We employed a reverse shell, i.e., *reverse_tcp*, from the Metasploit framework [177] to launch ROP exploits on Internet Explorer 8. Note that we uninstalled the Microsoft updates KB2744842 and KB2799329 in order to successfully launch the ROP exploits.

Our subjects include three benchmarks: 10 spraying-based and 1 non-spraying-based real-world ROP exploits from the Metasploit framework [177] as listed in Table 6.2, 50 synthetically generated ROP exploit variants, and 1000 benign websites that are the top websites in the ranking of Alexa [180]. Among the 10 spraying-based exploits, 3 of them use custom allocators, while 7 of them use system allocators to realize spraying. Therefore, the ROP exploits we choose for our

Table 6.2: ROP Exploits from the Metasploit Framework

#	Description	CVEs
1.	Adobe Flash Casi32 Int. Ovf.	2014-0569
2.	Adobe Flash Domain Memory UAF	2015-0359
3.	IE COALineDashStyleArray Int. Ovf.	2013-2551
4.	IE CButton UAF	2012-4792
5.	IE Fixed Table Col Span Heap Ovf.	2012-1876
6.	IE Execommand UAF	2012-4969
7.	IE CDisplayPointer UAF	2013-3897
8.	IE Same ID property	2012-1875
9.	IE Option Element UAF	2011-1996
10.	Adobe Flash Kern Parsing Int. Ovf.	2012-1535
11.	IE CGenericElement UAF (non-spraying)	2013-1347

experiments are representative for the entire spectrum of ROP exploits as described in Section 6.2.1. Note that, the number of ROP exploit samples is small in our experiments, as is similarly done in the literature [135, 161, 167]. The reason is that the number of publicly available working exploits is small and it is also challenging to make the exploits work in a given environment.

Using the three benchmarks, we aim to answer the following research questions through the experiments.

Q1. How is the generality of the observed behavioral patterns of ROP exploits and spraying attacks?

Q2. How are the parameters (i.e., the sampling rate s_l and s_h and the time window w_{hs} and w_{rop}) tuned with respect to detection accuracy and performance overhead?

Q3. What is the accuracy of our framework?

Q4. What improvement can be achieved by our framework over the state-of-the-art techniques?

6.6.1 Generality of ROP Exploit and Spraying Attack Patterns

We used the first five ROP exploits in Table 6.2 to empirically specify the patterns of ROP exploits and spraying attacks (see Section 6.4.1.1 and 6.4.2.1); and launched the remaining ROP exploits in Table 6.2 to evaluate the generality of the patterns.

Fig. 6.8 and 6.9 respectively show the value of the metrics that are used to model the behaviors of ROP exploits and spraying attacks over the execution time. For clarity, we only show the time segments that contain the ROP exploit or spraying attack. These metrics are summarized from the first five ROP exploits in Table 6.2, i.e., the first five rows in Fig. 6.8 and 6.9. We can clearly see that the remaining exploits in Table 6.2 also exhibit similar behaviors with respect to these metrics, as highlighted by red rectangles. In particular, during the ROP exploit, *Ret_Miss/Ret*, *Ret/Ins*, *ITLB_Miss/Ins* and *LLC_Miss/Ins* all have a large spike at the same time. On the other hand, during the spraying attack, *Br/Ins*, *Load_Uops/Uops* and *Load_Uops_L1_Hit/Uops* all keep almost constant despite some small spikes because of the noise from other processes. As discussed in Section 6.5, such noise is caused as our current implementation cannot separate the performance counter events from other processes during context switches. However, it can be reduced by employing the sliding window algorithm to compute the consistency.

Summary: these observations from Fig. 6.8 and 6.9 positively answer Q1 that the behavioral patterns can still hold for different ROP exploits and spraying attacks, and thus can be used to model the behavior of ROP exploits and spraying attacks.

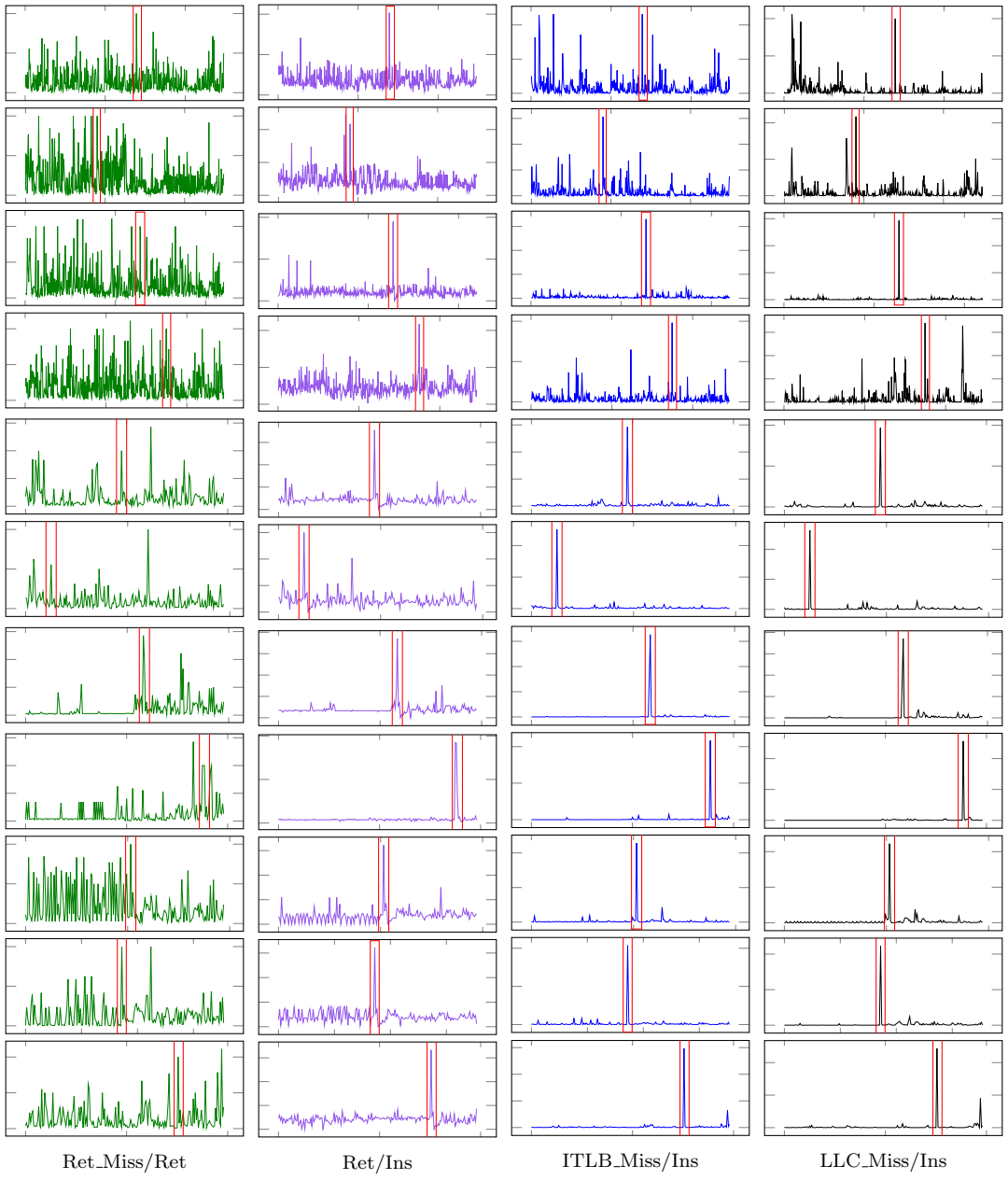


FIGURE 6.8: The ROP Exploit Behavior of the 11 ROP Exploits over the Execution Time

6.6.2 Accuracy and Overhead of ROP-Only Defense Approach

We tuned the sampling rate (s_h) for our ROP-only defense approach w.r.t. the detection accuracy and performance overhead using the 11 ROP exploits and 1000 benign websites. Fig. 6.10 reports the number of detected exploits with respect to

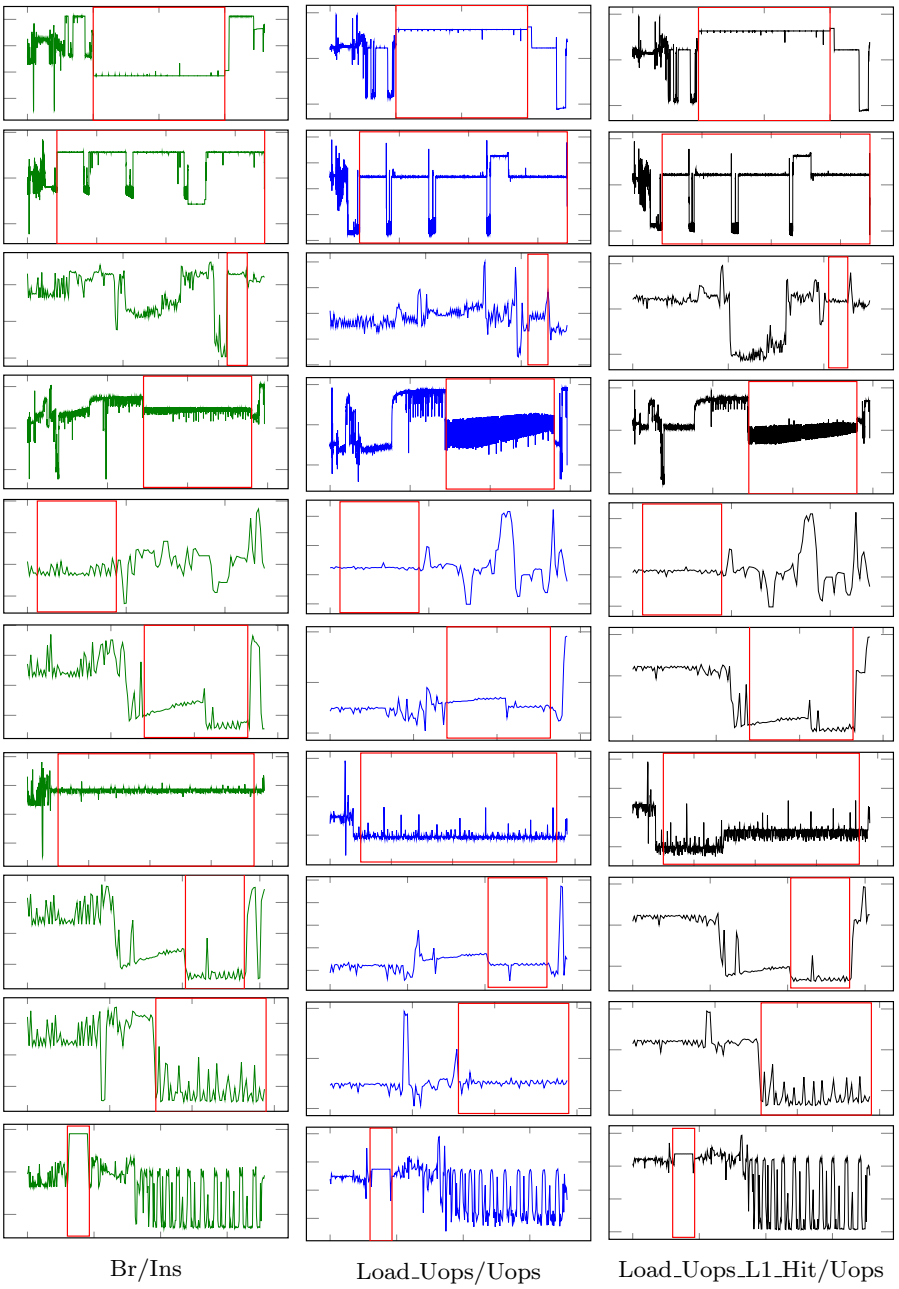


FIGURE 6.9: The Spraying Attack Behavior of the 10 Spraying-Based ROP Exploits over the Execution Time

the sampling rate in terms of return misses (i.e., at every N return misses). We can observe that the detection rate is high at high sampling frequency and it decreases with the decreasing of the sampling frequency. This is because when a large value for N is used, the small-length ROP chain cannot be clearly differentiated from the normal benign operations.

To estimate the corresponding performance overhead, we run top 50 Alexa-ranked

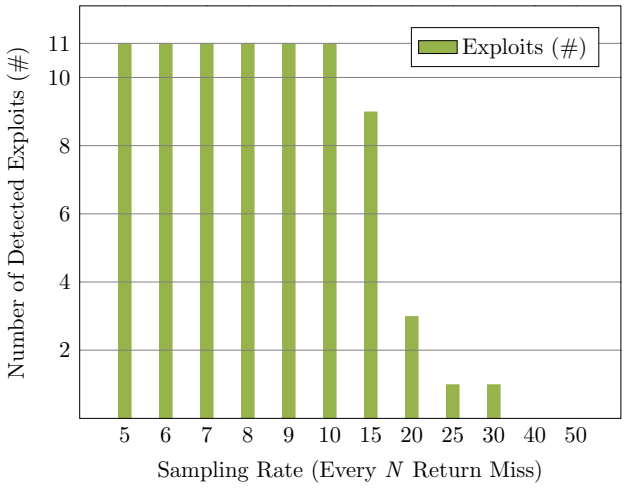


FIGURE 6.10: Detected Exploits w.r.t. Sampling Rate

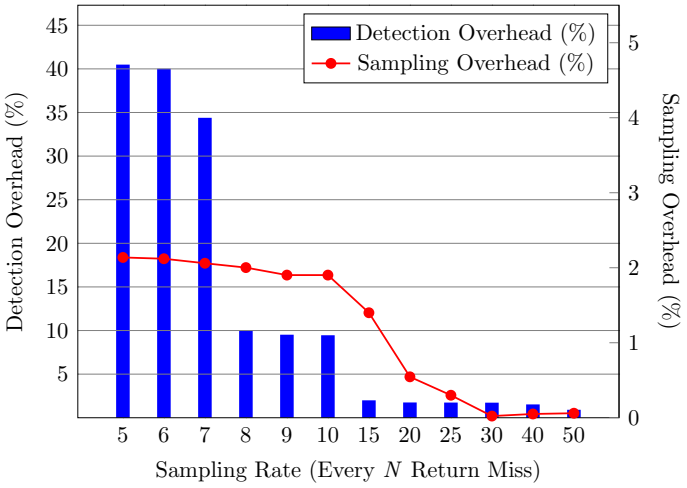


FIGURE 6.11: Performance Overhead w.r.t. Sampling Rate

websites at different sampling rates of return misses. We choose benign websites for performance estimation because exploits run for short time and occur rarely, whereas the benign website can run for a longer time. Since our runtime defense approach must be activated all the time, it is better to estimate the overhead by running large number of websites, as is similarly done in [161, 162]. In Fig. 6.11, we show the performance overhead that is incurred due to sampling at different intervals (by line graph) and overall detection process (by bar graph). It can be seen that the performance overhead due to sampling at high sampling frequency is much higher than that at low sampling frequency. Generally, the performance overhead due to sampling is at a low level, e.g., only around 2% even with the highest sampling frequency. On the other hand, the performance overhead due

to overall detection process is significantly high at the highest sampling frequency (40% when $N = 5$), which decreases at low sampling frequency. Thus, in our approach, we find that, as depicted in Fig. 6.10, 10 is an appropriate value of return misses for the sampling, i.e., hardware events are sampled at every 10 return misses to detect ROP chains. At this sampling rate, we observe that the overall performance overhead is around 11%.

Compared to the previous approach [167] that samples at 512K instructions, we observe that our sampling overhead (2%) is slightly higher than their approach (1.5%). However, their ROP exploit detection accuracy is $< 70\%$, while our approach has 100% detection rate. In terms of overall detection overhead, we cannot compare with their approach, because they performed only offline analysis and had no results for online detection. However, we believe that the overall detection overhead caused by their approach must be significantly higher than our approach (which is only around 9%). This is because, they used machine learning-based approach to perform the detection that involves a lot of computations, finally resulting into significant overhead during runtime. Comparatively, our approach is lightweight, involving only several instructions and thus having less overhead during runtime. Therefore, the ROP-only defense approach, with 11% performance overhead and high detection accuracy, is already a significant achievement compared to previous techniques.

Summary: based on our study on real-world ROP exploits, we set the high sampling rate (s_h) for ROP exploits to every 10 return misses, which answers Q2. With this configuration, our ROP-only defense approach can detect all the 11 ROP exploits while keeping the performance overhead around 11%.

6.6.3 Accuracy and Overhead of Self-Adaptive Defense Approach

Our self-adaptive defense approach relies on several parameters: the two sampling rate s_l and s_h to monitor the hardware events for spraying attacks and ROP

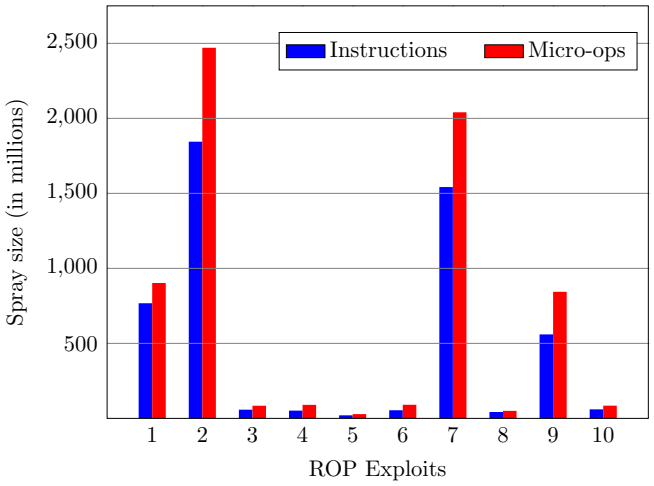


FIGURE 6.12: Spray Size of the 10 Spraying-Based ROP Exploits

exploits, and the two time window w_{hs} and w_{rop} to detect spraying attacks and ROP exploits. Since s_h was empirically set in Section 6.6.2, here we empirically set the other three parameters w.r.t. the detection accuracy and performance overhead.

The Time Window (w_{hs}) for Spraying Attacks. Spraying attacks usually involve millions of instructions. Fig. 6.12 shows the size of the sprays of the 10 spraying-based ROP exploits in Table 6.2 in millions of instructions and microinstructions. This ROP exploits benchmark demonstrates that, the size of a spray varies with the ROP exploit, and a spray can be observed from 30 to 2500 million microinstructions. In our approach, we safely take 20 million microinstructions as the time window for spraying attacks, i.e., if the summarized metrics are all consistent for at least 20 million microinstructions, we suspect that the pattern of spraying attack is detected and then switch to ROP detection. This time window may cause false positives for spraying attacks, but we have a further step to detect ROP chains. We use microinstruction as the unit for time windows and sampling rates as it is the unit of execution in CPU pipeline.

The Low Sampling Rate (s_l) for Spraying Attacks and Time Window (w_{rop}) for ROP Exploits. As shown in Fig. 6.12, spraying attacks occur over millions of microinstructions; and the lowest spray size is 28 million microinstructions. Thus, we explore the low sampling rate (s_l) for spraying attacks by varying it from every

TABLE 6.3: Detected Exploits w.r.t. s_l and w_{rop}

s_l	w_{rop}					
	1000	2000	3000	4000	5000	6000
256K	3	7	5	6	9	5
512K	5	4	4	7	8	7
1024K	5	4	8	6	9	7
2048K	6	8	7	8	8	7
3072K	5	7	7	7	7	7
4096K	3	9	9	10	10	8
5120K	4	7	7	10	7	10
6144K	6	6	8	**10**	9	9
8192K	5	5	8	9	8	8

256K to every 8192K microinstructions. For each fixed s_l, we also vary the time window (w_{rop}) for ROP exploits from 1000 to 6000 sampling intervals. Here s_h is set to every 10 return misses, and w_{hs} is set to 20 million microinstructions.

Table 6.3 shows the number of detected ROP exploits from the 10 ROP exploits, for each configuration of s_l and w_{rop}. We can see that, if s_l is low (e.g., every 8192K microinstructions), our approach may miss the ROP chains after switching to ROP detection. This is because a ROP chain has only hundreds of microinstructions, while s_l is every millions of microinstructions; i.e., the ROP chain might already occur in the last sampling interval of spraying attack. On the other hand, if s_l is high (e.g., every 256K microinstructions), we need to have a large w_{rop} for ROP detection, since the ROP chain will be executed for several sampling intervals.

As shown in Table 6.3, a large w_{rop} does not always lead to a high exploit detection rate. For example, when s_l is at every 6144K microinstructions, more exploits are detected when w_{rop} is 4000 sampling intervals than it is 5000 sampling intervals. This is because the size of the spray in some ROP exploits is extremely large, where several switches between spraying attack detection and ROP detection happen, and the large w_{rop} may make the ROP chain executed during the spraying attack detection. As a result, the ROP chain is missed. Fig. 6.13 depicts such a scenario,

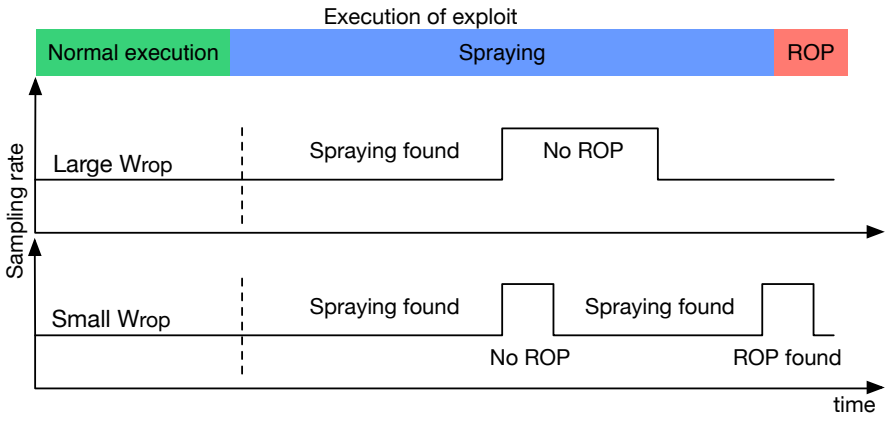


FIGURE 6.13: Impact of w_{rop} on ROP Detection

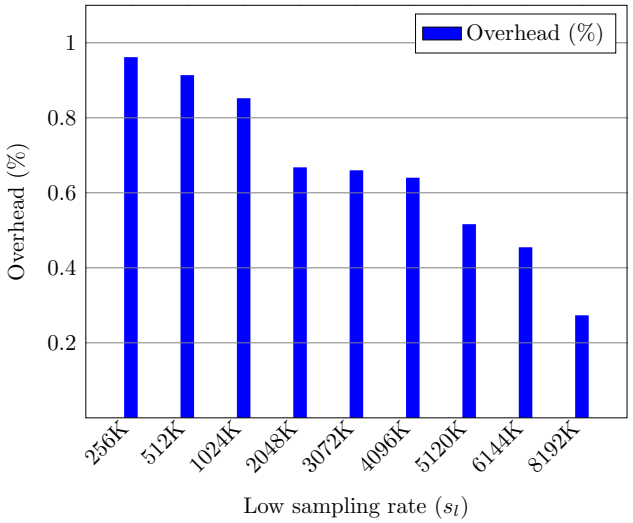


FIGURE 6.14: Overhead w.r.t. s_l (w_{rop} is 4000 sampling intervals)

where the ROP chain is missed with large w_{rop} as our approach switches back to the spraying attack detection when the ROP chain is executed, and a small w_{rop} can detect the ROP chain.

For performance overhead, thanks to our return miss-based sampling technique and our self-adaptive sampling technique, the performance overhead on the top 50 Alexa-ranked websites is 1%, as shown in Fig. 6.14 and 6.15 w.r.t. varying s_l and w_{rop}.

Summary: based on our empirical study on real-world ROP exploits, we set the low sampling rate (s_l) for spraying attacks to every 6144K microinstructions, the high sampling rate (s_h) for ROP exploits to every 10 return misses, the time window (w_{hs}) for spraying attacks to 20 million microinstructions, and the time

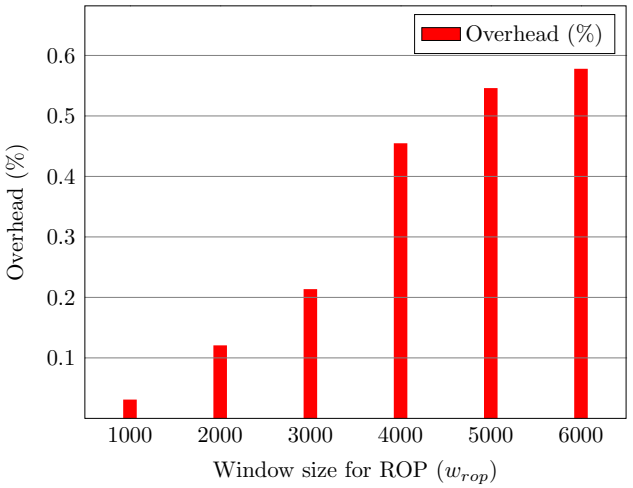


FIGURE 6.15: Overhead w.r.t. w_{rop} (s_l is every 6144K uops)

window (w_{rop}) for ROP exploits to 4000 sampling intervals, which answers Q2. With this parameter configuration, our approach can detect all the 10 ROP exploits while keeping the performance overhead under 1%. Compared to our ROP-only defense approach, the self-adaptive defense approach significantly reduces the performance overhead by maintaining the high detection accuracy.

6.6.4 Accuracy: False Negative and False Positive

To evaluate the false negative rate of ROPSentry, we ran the two approaches on 50 synthetically generated ROP exploits, which are variants of the published exploits presented in Table 6.2. Since the Metasploit framework does not allow the change of spraying attack and ROP chain, we manually created the variants of exploits by varying the spray memory allocation loops and shellcode size, following the same procedure as proposed in [162]. Specifically, Shellcode changes include addition of NOP sled to shellcode, addition of dummy instruction at the end of shellcode, and repetition of the original shellcode. Both ROP-only and self-adaptive defense approaches can detect all the 50 exploit variants, which indicated the generality of the patterns of ROP exploit and spraying attack and the effectiveness of our framework in detecting ROP attacks.

Table 6.4: False Positives in 1000 Benign Websites

Policy	1	2	3	4	1,2	1,2,3	1,2,4	1,2,3,4
F.P.	948	918	149	168	117	0	0	0

We also evaluate the false positive of our framework by running the top 1000 websites, ranked by Alexa [180], on Internet Explorer 8. We loaded the first page of the website, and observed the performance counter values for spraying attacks and ROP exploits by our framework. To show the ROP exploit detection capability of different policies (see Section 6.4.1.1), we analyze the false positive with different combinations of the policies. As shown in Table 6.4, one individual policy leads to extremely high false positives, which is also evidenced in Fig. 6.8 by the many spikes during the normal executions. However, by combining policy 1 and 2 that are based on return misses and return instructions, the false positive rate is significantly reduced to 11%. After we further combine policy 3 (ITLB misses) and 4 (LLC misses) respectively, there is no false positive. The reason is that, (1) the first two policies might also hold true in certain conditions, e.g., during exceptions; however, such cases do not have high ITLB misses and LLC misses; and (2) policy 4 is closely related to policy 3 as high ITLB misses often lead to high LLC misses. Therefore, when all the policies are combined, we have zero false positive. Note that without using LLC miss, we can still achieve zero false positives (shown in Table 6.4). We prefer to keep LLC miss because we have the flexibility of four programmable performance counters, and it does not introduce significant overhead to our approach. Note that both ROP-only and self-adaptive defense approaches have the same result as they use the same pattern for detecting ROP exploits.

Summary: based on these observations, we can positively answer Q3 that, the false negative and false positive of our framework for detecting ROP exploits are very low.

6.6.5 Comparison with the-State-of-the-Art Tools

Enhanced Mitigation Experience Toolkit (EMET) [168] is the-state-of-art tool developed by Microsoft that aims to make exploitation harder. Therefore, we compared our approach with EMET. EMET has several mitigation techniques that make it harder to perform the exploits; however, they are not sufficient to completely protect from the exploits. Specifically, EMET offers one spraying attack mitigation technique, and five ROP mitigation techniques (StackPivot, SimExecFlow, Caller Check, MemProt, and LoadLibrary [181]).

Table 6.5 compares the exploit detection capability of our approach against EMET v5.5. As observed from our experiments, EMET techniques, including Heap Spray, SimExecFlow, Caller Check, MemProt and LoadLibrary are ineffective against most of the exploits. The most effective technique in EMET is StackPivot; however, as it performs the check at each critical API, it can be easily defeated by using the hook hopping technique as explained in Section 6.2.3. Differently, our approach can detect all the 11 ROP exploits using low-level hardware events. Here we do not compare their performance overhead because EMET is not open-sourced and we cannot get the detailed performance information for the comparisons.

We also attempted to compare our approach with the existing HPC-based defense techniques in terms of the detection capability and performance overhead. However, to the best of our knowledge, none of them are publicly available. Instead, we compared our approach with the technique in [167] based on the data shown in their paper. Since the performance overhead of HPC-based mitigation is mainly determined by two factors: the sampling frequency and the data analysis technique, we conduct the comparison by these two factors. First, our sampling rate is around every 6144K microinstruction at low sampling rate, while their sampling rate is every 512K instructions. Second, we use light-weight techniques to detect spraying attacks and ROP exploits, which do not involve large amount of computation and therefore, the overall performance overhead is kept under 1%, as discussed in Section 6.6.3. However, they use heavy machine learning techniques to analyze the data at each PMI, which can cause high overhead as it involves

large amount of computation. In comparison, their sampling at 512K itself has performance overhead of 1.5% and on top of it, the ROP detection rate at this sampling rate is just 70%. As stated earlier (Section 6.6.2), the overhead will be significantly high if the detection part is performed during runtime, because of the machine learning based implementation.

Besides, we compared our framework with the most recent technique Graffiti [161], which aims to detect spraying attacks. Their approach can only detect system allocator-based spraying attacks, but cannot handle custom allocator-based spraying attacks. We also attempted to run our framework on the 6 exploits used by them. All exploits date back to 2011 or before. We could only run two exploits: CVE-2011-1996 and CVE-2010-2883; and our framework can detect both of them. We could not run the other 4 exploits as they can run successfully only on Windows XP, which is an old platform. However, our framework supports the more advanced Windows 7 platform. In terms of execution overhead, we have around 11% overhead using our ROP-only approach and 1% overhead using our self-adaptive approach, while Graffiti has 23% overhead for detecting spraying attacks.

Summary: the comparisons with EMET techniques and the approach in [167] and [161] answers Q5 that our framework can significantly improve performance overhead without sacrificing the detection capability.

6.7 Discussion

Our approach has some limitations. First, our ROP exploit detection technique is based on the abnormal behavior in terms of return misses, return instructions, ITLB misses, and LLC misses. Among them, return miss is one critical evidence in the current ROP exploits. It can be defeated by the following chain of gadgets: interrupt return gadgets [182], indirect call-gadgets-ret (COOP attack [183]), and indirect call-gadgets-indirect jmp chains. However, these are more theoretical exploits and are not common in real world; and our approach can be enhanced

[1]IE crashed but no EMET warning message

Table 6.5: Comparison with EMET Techniques

Exp.	Heap Spray	Stack-Pivot	Sim-Exec.	Caller Check	Mem-Prot	Load-Lib.	Ours
1	×	✓	×	✓	×	×	✓
2	×	✓	×	✓	×	×	✓
3	×	✓	✓	✓	×	×	✓
4	✓	✓	×	✓	×	×	✓
5	×	✓[1]	×	×	×	×	✓
6	✓	✓	✓	×	×	×	✓
7	✓	✓	✓	✓	×	×	✓
8	×	✓	×	×	×	×	✓
9	✓	✓	✓	✓	×	×	✓
10	×	✓	✓	✓	×	×	✓
11	×	✓	✓	×	×	×	✓

with more policies to consider indirect call and jmp. For example, we can use indirect jmp misses and indirect call misses to detect such theoretical exploits. To the best of our knowledge, no COOP-based ROP exploits are publicly available till date, and thus we cannot incorporate such type of attacks in this work.

Second, the patterns to model the behavior of spraying attacks and ROP exploits are empirically determined by only a small number of real-world ROP exploits, which may hinder the effectiveness of our approach although our experiments have shown promising results. On one hand, the number of publicly available working ROP exploits is small (similar experience and experiments have been shown in other work [135, 167]). On the other hand, our approach can be easily extended to integrate new patterns with the emerging of new ROP exploits. Similarly, the four parameters (two sampling rates and two time windows) in our self-adaptive sampling techniques are also empirically determined based on a small number of real-world ROP exploits. As a result, our approach might miss ROP exploits. One possible scenario is that, after performing the spray, the attacker waits until the ROP detection duration (w_{rop}) has been elapsed (by performing some normal code

execution) and the state returns to normal. During the normal state, the attacker can choose to trigger the ROP payload to bypass the detection. However, the complexity of performing a successful ROP exploit will highly increase. Because the spraying in the heap memory might have been destroyed due to normal code allocation in the heap, making the spraying ineffective. This will certainly reduce the chances of performing a successful exploit. One possible solution is to dynamically determine the sampling rates and time windows according to the behavior of spraying attacks and ROP exploits.

Last, our current implementation is still limited by the number of programmable performance counters in current processor although we propose the self-adaptive sampling technique to monitor different sets of hardware events for spraying attacks and ROP exploits. However, the next generation processors may have 8 or 16 programmable performance counters to monitor more hardware events, which can be leveraged to model the behavior of spraying attacks and ROP exploits more precisely.

We also want to discuss whether an attacker can create a mimicry attack by knowing the internal details of our ROPSentry framework. The key problem with this attempt is that the attacker has no direct control to manipulate these hardware events, while still being able to perform the attack successfully. While one may be able to control the architectural features, such as branch instructions and return instructions, it is almost impossible for the attacker to manipulate all the low-level non-architectural features such as *LLC_Miss*, *ITLB_Miss*, *Load_Uops* and *Load_Uops_L1_hit*, as these events cannot be directly controlled. It may be possible that an attacker can fool one of the hardware events, however, it is almost impossible to manipulate multiple hardware events simultaneously, while still performing a successful attack. Thus, we believe that our framework is robust against mimicry attacks.

6.8 Conclusions

We proposed and implemented a defense framework, called *ROPSentry*, to detect ROP exploits at runtime in this work. The proposed framework provides a complete solution to handle the entire spectrum of ROP attacks. Our ROP-only defense approach leverages hardware performance counters to monitor the abnormal behavior of hardware events that can be used to model the behavior of ROP exploits. Besides, we proposed a self-adaptive defense approach to dynamically sample the hardware events, to additionally detect spraying attack (which is a common payload delivery mechanism), such that the performance overhead can be reduced to a very low level. We evaluated the proposed approach on 11 real-world and 50 synthetically generated ROP exploits and 1000 benign websites, which shows promising results that our approach can effectively detect ROP exploits at runtime with low performance overhead and low false positive rate.

In the future, we plan to investigate the possibility to dynamically determine the key parameters in our framework. In addition, we hope to reduce the noise from other processes to precisely capture the behavior of spraying attacks and ROP exploits. Besides, our self-adaptive defense approach currently cannot confirm spraying attacks, since we do not look into the memory content. We plan to extend our ROPSentry framework to confirm spraying attacks by investigating into the memory content through hardware events such as DTLB-store-misses.

7

Conclusions and Future Research

The number of security breaches has increased recently and will continue to grow in near future because of the evolution of sophisticated attacks by the motivated adversaries, either for financial gains or for causing damage to the victim. Malware and exploits are two critical issues of software security. This thesis has proposed several hardware-based approaches and architectures for runtime security at the pre-infection (i.e., at exploitation stage) and the post-infection stages (i.e., against the malware).

We started with designing a hardware-based malware detection engine, using machine learning approach in FPGA, to protect against malware at runtime. We showed that the proposed approach is scalable and can effectively detect unknown malware samples based on the trained features. We then presented an approach to capture the attack behavior of malware, which has the potential to detect zero-day

malware. Implemented in hardware, our architecture offers a real-time detection with low performance and resource overhead. Moreover, it is resilient to sophisticated malware evasive techniques.

We have also shown that the hardware-enhanced architecture can offer protection against the critical exploits at runtime, which can eliminate the threats at the pre-infection stage, even before the malware is downloaded. Our solution offers an effective defense against exploits with high accuracy and low false positives. Besides, the defense mechanisms are implemented in the hardware that cannot be bypassed or disabled by exploit or malware. We also showed that with proper support of hardware design, our architecture offers low power and energy consumption, which is suitable for online defense.

This chapter concludes our contributions that are described in this thesis, compares them and finally, outlines the direction for future research.

7.1 Summary of Contributions

We highlight the key contributions of this thesis in this section.

7.1.1 Online Malware Detection Using Machine Learning Approach

In recent years, there has been an increasing number of malicious attacks. The mainstream solutions have been ineffective against the sophisticated techniques used by malware to bypass them. As shown in Chapter 3, we designed GuardOL, a hardware-enhanced architecture to perform online malware detection. GuardOL is a combined approach using processor and FPGA. It aims to capture the malicious behavior (i.e., high-level semantics) of malware. We proposed the frequency-centric model for feature construction using system call patterns of known malware

and benign samples. We then developed a machine learning approach (using multilayer perceptron) in FPGA to train a classifier using these features. At runtime, the trained classifier is used to classify the unknown samples as malware or benign, with early prediction. We showed that our solution can achieve high classification accuracy, fast detection, low power consumption and flexibility for easy functionality upgrade to adapt to new malware samples.

7.1.2 Online Malware Defense using Attack Behavior Model

In Chapter 4, we presented a DFA-based online malware detection approach, called *Malguard*, which can effectively detect malware variants and has the potential to detect zero-day malware. Malguard learns the attack model of malware in the form of DFA. At runtime, it checks whether a program's execution contains the malicious behavior specified in the DFA. We also introduced a frequency parameter to model the repetitive malicious behavior. The evaluation shows that Malguard can recognize malware variants of the same family with the potential to catch zero-day attacks, and has a high detection rate with no false positive. In addition, it facilitates real-time malware detection with low performance and memory overhead.

7.1.3 Control Flow Integrity Approach against Runtime Memory Attacks

Runtime attacks on memory, such as buffer overflow based stack smashing and code reuse attacks (ROP and JOP), are common in embedded systems. In Chapter 5, we presented a BB-CFI approach to enforce a fine-grained CFI policy at a basic block level to defend against these attacks. Based on the offline profiling of the program, our control flow checker verifies the target address of the control flow instructions (including `call, ret` and `jmp`) at runtime. We also presented the architecture of control flow checker (CFC), which monitors the program execution during runtime to enforce BB-CFI. The CFC design is prototyped in FPGA. Our method does

not require the modification of the code (source or binary) or the Instruction Set Architecture (ISA). We demonstrated the effectiveness of our approach by evaluating against several benchmarks. The CFC implementation on FPGA shows <1% performance overhead and small dynamic power consumption of 78 mW, with a very small area footprint.

7.1.4 Defense against ROP Exploits using Hardware Performance Counters

Return-oriented programming (ROP) is the most dangerous and widely-used technique to exploit software vulnerabilities. However, the existing defense techniques either can be easily defeated by attackers, or require high performance overhead, and thus lack viability for real-world deployment. In Chapter 6, we demonstrated a novel defense approach to detect ROP attacks at runtime. Our approach builds on the observation that ROP programs, when executed, trigger different hardware events than normal programs generated by compilers. Therefore, we leverage hardware performance counters to track these hardware events so that we can capture the patterns of heap spray (i.e., a common ROP payload delivery mechanism) and ROP exploit. To reduce the performance overhead, we proposed a self-adaptive sampling technique to dynamically switch between the low and high sampling rate for fetching the hardware performance counter values. We conducted our evaluation on real-world exploits and benign websites, which shows promising results in detecting ROP attacks at runtime with a low performance overhead and a low false positive rate.

7.2 Comparison of Approaches

In this section, we compare our proposed approaches and discuss their pros and cons. We also highlight their applications in the real-world settings. Table 7.1 shows the comparison of our approaches in detail.

Table 7.1: Comparison of our approaches

	GuardOL	Malguard	BB-CFI	ROPSentry
Target attacks	General malware - virus, trojan, worm, flooder, rootkit, backdoor	General malware, variants and zero-day malware	Control flow attacks – stack smashing and code reuse attacks (ROP, JOP)	Heap spray and ROP attacks
Approach	Feature-based model using machine learning approach	Attack-behavior model in the form of DFA	Control flow integrity at basic block level	Performance monitoring unit (PMU) based approach
Information used	System calls and parameters	System calls and parameters	Branch address	Non-/architectural events
Assumption	Trusted system call libraries	Trusted system call libraries	Trusted profiling and program binary	PMU featured processor
Methodology	Features extraction to train machine learning classifier for runtime classification	Malware attack behavior modeling in the form of DFA using L* algorithm	Runtime verification of branch targets based on offline profiling of basic blocks	Return miss based and Self-adaptive sampling of performance counters
Architecture modification	Hardware-enhanced architecture of GuardOL	FPGA-based implementation of Malguard	Hardware-enhanced architecture of Control flow checker	No hardware modification, uses PMU logger and analyzer
Implementation framework	Multi2sim CPU simulator (x86-32-bit), FPGA	Multi2sim CPU simulator (x86-32-bit), FPGA	Valgrind BBV, Multi2sim CPU simulator (x86-32-bit), FPGA	Intel Xeon Haswell, Windows 7
Benchmarks	Real world linux malware and benign programs	Real world linux malware and benign programs	Shellcode attacks, RIPE, SPEC CPU 2006, Mibench	Real world and synthetic exploits and benign websites
Accuracy	TP - 98%, FP - <3%	No FP	High accuracy, No FP	TP - 100%, No FP
Overhead	No performance overhead, Area <1%, Power <1% (86mW)	No performance overhead, Area <1%, Power <1% (86mW)	Performance <1%, Area <0.02%, Power <78mW	Performance <1%
Advantages	Early prediction, faster training and detection, low runtime overhead, scalable	Early prediction, attack behavior recognition and malware variants detection	Indirect control flow reduction (>99%), gadget reduction (>99%), low overhead (<1%)	Practical, scalable, effective, low overhead (<1%)
Limitations	Cannot detect malware variants and zero-day malware	Malware attack behaviors should be known	Offline profiling must cover all the valid control flow paths	Analyzer can be disabled if OS is compromised
Applications	Embedded systems, Mobile platforms	Mobile platforms, IoT and embedded devices	IoT and embedded devices	Any modern computing platform - desktop or mobile systems

7.2.1 Strength and Weakness

The first two works – GuardOL and Malguard – focus on general malware detection using hardware-enhanced architecture. They extract the high-level behavior of the program using system calls and their parameters at the runtime in order to build

the model. The advantage of these approaches is that they can detect malware at the early stage of execution. GuardOL is a scalable approach and offers faster training and detection with low overhead. It learns the malicious features from the known malware samples, however, it cannot model the attack behavior of malware. Thus, it can be defeated by modifying the known malicious behavior.

On the other hand, Malguard learns the attack model of malware in the form of DFA using the L* algorithm. Hence, it can detect malware variants and potentially zero-day malware. This is because malware variants use similar attacking behavior techniques to achieve their malicious intents. The limitation of this approach is that the attack behavior of a particular malware should be known during the learning phase. This limits the applicability of this approach. However, given the fact that malware classification has been done by a large number of tools [97], this limitation can be solved. Also, the DFA-based detection can run faster than machine learning, as the computation involved in DFA verification is less compared to the machine learning classifier.

BB-CFI and ROPSentry aim to defend against the popular exploits that are commonly used by the adversary for penetration of malware into the system. These approaches aim to protect the pre-infection stage, thus preventing from malware downloading. BB-CFI implements a strict CFI approach to monitor the exploits that deviate the normal control flow of the program. It verifies the branch address of the control flow instructions (e.g., `call, ret, jmp instruction`) at runtime to detect the control flow attacks. However, this approach requires the offline profiling of the programs under the trusted environment. This approach is suitable for certain embedded applications that run a limited set of programs, which have defined functionalities, e.g., ATM machine. For complicated applications, this approach may give high false positives due to the incomplete profiling information. This is because for complex programs complete profiling of all the program paths may not be feasible.

Our final work, ROPSentry, proposes a more practical, effective and highly scalable technique that uses CPU feature of commodity processors to detect ROP exploits.

It monitors architectural and non-architectural events (such as instructions, cycles, branch and cache accesses, load/store operations) using performance counters to model the ROP exploits. Unlike BB-CFI, it does not require offline profiling of applications. Thus, it can detect ROP exploits on any unknown programs that run on the system. On top of it, this approach does not require any modification in CPU architecture and is highly scalable as it can be implemented on any desktop or mobile platforms. Since the overhead is low (approximately 1%), it can be easily adapted to the real-world settings. The only limitation is that if the OS kernel is compromised, the detection mechanism can be disabled. Current settings only aim at the practical ROP exploits, however, by monitoring more events, it can be extended to defend other exploits.

7.2.2 Information Used for Attack Detection

Table 7.1 highlights the use of different information that is leveraged by our approaches for detecting malware and exploits. GuardOL and Malguard use system calls and their parameters, to obtain high-level semantics of the malicious behavior. BB-CFI uses branch addresses of the control flow instructions (e.g., `call, ret, jmp`) to verify the control flow of the program during runtime. On the other hand, ROPSentry monitors architectural and non-architectural events – such as instructions, cycles, branch and cache accesses, load/store operations – using performance counters to model the heap spray and ROP exploits.

7.2.3 Target Applications

GuardOL is a scalable approach with low resource requirements, therefore, it is suitable for embedded applications and mobile platforms. On the other hand, Malguard is a lightweight approach compared with GuardOL, as the learning is performed offline using a software approach. Hence, it is suitable even for the smaller IoT applications, as the overhead is less compared to GuardOL. BB-CFI

approach requires offline profiling of the programs that run on the system. Therefore, it is more suited for the applications that have limited functionalities, e.g., ATM, medical devices, home appliances, and other IoT applications. ROPSentry is more robust than BB-CFI and does not require profiling. On top of it, it does not need any modification in hardware and can be implemented on any modern processor systems. The online detection of exploits during runtime with low overhead makes it suitable for any applications, ranging from desktop to mobile, such as smartphones, tablets.

7.3 Future Research

The research presented in this thesis aims at advanced architectures and solutions for online defense. We have identified numerous possible extensions to the work presented in this thesis which can be further explored as future research.

7.3.1 Malware Detection Engine for Multicore Systems

In Chapter 3 and 4, we proposed hardware-based malware detection engine. The current designs are focused on the single-core processor systems. Since multi-core processors are mostly used nowadays, we can extend our proposed design to support runtime security in the multi-core environment. There are several new problems that need to be addressed when considering the multi-core architecture.

First, the reconfigurable architecture is connected close to the single processor core through a special track to achieve fast data collection and analysis. In the multi-core systems, as malware can run on multiple processor cores, the data from multiple cores have to be aggregated first to form a complete feature set to capture the semantics of malware. Hence, the security module needs to connect to the bus to facilitate the communication with all the processor cores. The data can be processed to build the feature, either centrally on a common core or individually at the each core to reduce the communication volume. Second, the data obtained

from different cores must be identified, based on the application from which it is obtained (using thread/process ID). In addition, security analysis methods may require to observe the data in the program execution order. Third, the data collection on the bus must be adjusted to incur low performance overhead to the user applications.

7.3.2 BB-CFI Policy Enforcement without Offline Profiling

BB-CFI presented in Chapter 5 enforces the fine-grained CFI policy at runtime by identifying the basic blocks during offline profiling. But obtaining the complete legitimate program paths by profiling the binary is a challenging task. The future research can explore the possibility of identifying the basic blocks at runtime using certain heuristics. A basic block is a set of instructions that has a single entry point and a single exit point. Thus, most of the basic blocks terminate in a branch instruction. Our BB-CFI approach allows that the target address of the branch instruction must be the starting address of a basic block. This means that the instruction before the branch target address must be a branch instruction because it must be the last instruction of the other basic blocks. This assumption might have some exceptions, as compiler sometimes add the NOP instructions at the end of the basic blocks. To consider such exceptional cases, BB-CFI policies can be relaxed to reduce the false positives. Thus, the future research in this direction can avoid the need for offline profiling and can still enforce CFI at the basic block level.

7.3.3 Modeling Advanced Exploits using CPU Features

Chapter 6 presents an online defense approach based on CPU performance counters, to detect widely used ROP exploits. As discussed in Section 6.7, our approach must be enhanced to defend the advanced theoretical exploits which may be more

practical in future, such as exploits that use – interrupt return gadgets [182], indirect call-gadgets-ret (COOP attack [183]) and indirect call-gadgets-indirect jmp chains, and JOP attacks. We need to investigate and develop more policies to consider such attacks, which mainly use indirect calls and jumps. For example, we can develop policies based on indirect call/jump misses to detect such exploits. In addition, other CPU features can be incorporated with PMU to model more exploitation techniques. Furthermore, the detection approach can also be implemented in the hardware, which cannot be disabled or bypassed.

7.4 Summary

This thesis has contributed novel approaches for hardware-assisted security against runtime attacks for computing systems. The focus was on leveraging hardware architectures and features to provide runtime security in a manner that has minimal impact on performance and existing architecture. These approaches can be ported to the modern computing systems in order to enhance their security against the critical security threats. We believe that our presented approaches, techniques, and enhancements will be beneficial for the mobile and embedded systems designers, as well as offering ideas in the general area of IoT systems. Having demonstrated the significant benefits of hardware-assisted security, we are hopeful that this work will encourage adoption of hardware-based security in future computing systems.

Bibliography

[1] "Symantec: Internet Security Threat Report 2016." [Online]. Available: https://www.symantec.com/security-center/threat-report

[2] "Mcafee Labs Threats Report March 2016," Tech. Rep., accessed April., 2016. [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf

[3] K. Lab, "The great bank robbery: Carbanak." [Online]. Available: http://www.kaspersky.com/about/news/virus/2015/Carbanak-cybergang-steals-1-bn-USD-from-100-financial-institutions-worldwide

[4] J. Radcliffe, "Hacking medical devices for fun and insulin: Breaking the human scada system," in *Black Hat Conference presentation slides*, vol. 2011, 2011.

[5] T. Chen, "Stuxnet, the real start of cyber warfare?[editor's note]," *Network, IEEE*, vol. 24, no. 6, pp. 2–3, 2010.

[6] (2014, December) Misfortune Cookie Vulnerability Exposes Millions of Routers. [Online]. Available: http://www.securityweek.com/misfortune-cookie-vulnerability-exposes-millions-routers

[7] "Atm jackpotting." [Online]. Available: http://www.wired.com/2010/07/atms-jackpotted/

[8] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, 2015.

[9] F. Mercês, "The brazilian underground market." [Online]. Available: http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-the-brazilian-underground-market.pdf

[10] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *Security & Privacy, IEEE*, vol. 9, no. 3, pp. 49–51, 2011.

[11] E. Chien, L. OMurchu, and N. Falliere, "W32. duqu: the precursor to the next stuxnet," in *Presented as part of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2012.

[12] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[13] "An unofficial analysis of the Retaliation Virus, Nov., 2014." [Online]. Available: http://vxheaven.org/lib/vrn01.html

[14] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: attacks, implications and opportunities," in *Proc. of HotMobile*, 2010, pp. 49–54.

[15] I. You and K. Yim, "Malware obfuscation techniques: A brief survey." in *Proc. of BWCCA*, 2010, pp. 297–300.

[16] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proc. of DSN*, 2008, pp. 177–186.

[17] J. Guilford, K. Yap, and V. Gopal, "Fast sha-256 implementations on intel architecture processors," 2012.

[18] "Intel® Software Guard Extensions Programming Reference." [Online]. Available: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

[19] "Crimeware Protection: 3rd Generation Intel® CoreTM vProTM Processors." [Online]. Available: http://www.intel.sg/content/dam/www/public/us/en/documents/white-papers/3rd-gen-core-vpro-security-paper.pdf

[20] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on.* IEEE, 2007, pp. 231–245.

[21] "Symantec: Security response. http://www.symantec.com/en/sg/security_response/publications/threatreport.jsp."

[22] E. H. Spafford, "The internet worm incident," in *Proc. ESEC*, vol. 89, 1991.

[23] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2014.

[24] Z. Bazrafshan, H. Hashemi, S. Fard, and A. Hamzeh, "A survey on heuristic malware detection techniques," in *2013 5th Conference on Information and Knowledge Technology (IKT)*, May 2013, pp. 113–120.

[25] G. Meng, Y. Liu, J. Zhang, A. Pokluda, and R. Boutaba, "Collaborative security: A survey and taxonomy," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 1, 2015.

[26] P. Gutmann, "The commercial malware industry," in *DEFCON conference*, 2007.

[27] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic Analysis of Malicious Code," *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, Aug. 2006. [Online]. Available: http://link.springer.com.ezlibproxy1.ntu.edu.sg/article/10.1007/s11416-006-0012-2

[28] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. of S&P*, 1996, pp. 120–128.

[29] G. Creech and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns," *IEEE Transactions on Computers*, vol. 63, no. 4, pp. 807–819, Apr. 2014.

[30] M. Alazab, S. Venkataraman, and P. Watters, "Towards understanding malware behaviour by the extraction of API calls," in *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, Jul. 2010, pp. 52–59.

[31] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 122–132. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336768

[32] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 399–412. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866353

[33] M. Chandramohan, H. B. K. Tan, L. C. Briand, L. K. Shar, and B. M. Padmanabhuni, "A scalable approach for malware detection through bounded feature space behavior modeling," in *Proc. of ASE*, 2013, pp. 312–322.

[34] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395, Oct. 2010.

[35] A. Liu, X. Jiang, J. Jin, F. Mao, and J. Chen, "Enhancing system-called-based intrusion detection with protocol context," in *SECURWARE 2011, The Fifth International Conference on Emerging Security Information, Systems and Technologies*, 2011, pp. 103–108.

[36] S. B. Mehdi, A. K. Tanwani, and M. Farooq, "Imad: in-execution malware analysis and detection," in *Proc. of GECCO*, 2009, pp. 1553–1560.

[37] J. Wang, Y. Xue, Y. Liu, and T. H. Tan, "Jsdc: A hybrid approach for javascript malware detection and classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 109–120.

[38] Y. Xue et al., "Detection and classification of malicious javascript via attack behavior modelling," in *Proc. of ISSTA*, 2015, pp. 48–59.

[39] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proc. of AISec*, 2013, pp. 45–54.

[40] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. of ISEC*, 2008, pp. 5–14.

[41] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 78–97.

[42] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in Computer Virology*, vol. 7, no. 4, pp. 233–245, Nov. 2011. [Online]. Available: http://link.springer.com.ezlibproxy1.ntu.edu.sg/article/10.1007/s11416-011-0151-y

[43] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic Modelling of Android Malware for Effective Malware Comprehension, Detection, and Classification." In The International Symposium on Software Testing and Analysis (ISSTA), 2016.

[44] A. Narayanan, G. Meng, L. Yang, J. Liu, and L. Chen, "Contextual weisfeiler-lehman graph kernel for malware detection," *arXiv preprint arXiv:1606.06369*, 2016.

[45] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, "Adaptive and scalable android malware detection through online learning," *arXiv preprint arXiv:1606.07150*, 2016.

[46] G. Meng, Y. Xue, C. Mahinthan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 365–376.

[47] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, "Effective and efficient malware detection at the end host." in *USENIX Security Symposium*, 2009, pp. 351–366.

[48] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 45–60.

[49] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, pp. 156–168, 2007.

[50] I. Santos, F. Brezo, J. Nieves, Y. K. Penya, B. Sanz, C. Laorden, and P. G. Bringas, "Idea: Opcode-sequence-based malware detection," in *Engineering Secure Software and Systems*. Springer, 2010, pp. 35–43.

[51] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Information Sciences*, vol. 231, pp. 64–82, 2013.

[52] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *Journal in Computer Virology*, vol. 8, no. 1-2, pp. 37–52, 2012.

[53] M. Zhang, A. Raghunathan, and N. K. Jha, "A defense framework against malware and vulnerability exploits," *International Journal of Information Security*, Mar. 2014. [Online]. Available: http://link.springer.com/10.1007/s10207-014-0233-1

[54] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," in *Proceedings of the*

15th ACM Conference on Computer and Communications Security, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 51–62. [Online]. Available: http://doi.acm.org/10.1145/1455770.1455779

[55] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999*, 1999, pp. 133–145.

[56] J. Cheng, S. H. Wong, H. Yang, and S. Lu, "Smartsiren: Virus detection and alert for smartphones," in *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '07. New York, NY, USA: ACM, 2007, pp. 258–271. [Online]. Available: http://doi.acm.org/10.1145/1247660.1247690

[57] A. Bose, X. Hu, K. G. Shin, and T. Park, "Behavioral detection of malware on mobile handsets," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. New York, NY, USA: ACM, 2008, pp. 225–238. [Online]. Available: http://doi.acm.org/10.1145/1378600.1378626

[58] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[59] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 1998.

[60] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security & Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[61] R. Gera, "Advances in format string exploitation," *Phrack Mag*, vol. 59, no. 7, p. 533, 2002.

[62] Blexim, "Basic Integer Overflows," *Phrack Magazine*, vol. 60(10), 2002. [Online]. Available: http://phrack.org/issues/60/10.html

[63] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[64] S. Designer, "return-to-libc" attack," *Bugtraq, Aug*, 1997.

[65] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. of CCS*, 2007, pp. 552–561.

[66] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM conference on Computer and communications security.* ACM, 2008, pp. 27–38.

[67] T. Kornau, "Return oriented programming for the arm architecture," Ph.D. dissertation, Master's thesis, Ruhr-Universität Bochum, 2010.

[68] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proc. of ASIACCS*, 2011, pp. 30–40.

[69] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security.* ACM, 2010, pp. 559–572.

[70] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.

[71] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. of SP*, 2014, pp. 575–589.

[72] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *Proc. of CCS*, 2005, pp. 340–353.

[73] C. Zhang et al., "Practical control flow integrity and randomization for binary executables," in *Proc. of Security and Privacy*, 2013, pp. 559–573.

[74] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. of ACSAC*, 2011, pp. 353–362.

[75] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *Proc. of DSN*, 2012, pp. 1–12.

[76] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *Proc. of ISCA*, 2012, pp. 94–105.

[77] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proc. of DAC*, 2014, pp. 1–6.

[78] S. Das, W. Zhang, and Y. Liu, "A fine-grained control flow integrity approach against runtime memory attacks for embedded systems."

[79] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz, "Control-flow integrity: Precision, security, and performance," *arXiv preprint arXiv:1602.04056*, 2016.

[80] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[81] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *Information Systems Security*. Springer, 2009, pp. 163–177.

[82] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[83] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. of ASIACCS*, 2011, pp. 40–51.

[84] I. Fratric, "Runtime prevention of return-oriented programming attacks," *University of Zagreb*, 2012.

[85] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 49–58.

[86] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic Integrity Measurement and Attestation," in *STC*, 2009, pp. 49–54.

[87] "Mcafee Labs Threats Report Quarter 3, 2014," Tech. Rep., accessed Jan., 2014. [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2014.pdf

[88] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Proc. of NDSS*, vol. 3, 2003, pp. 191–206.

[89] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[90] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the 1st India software engineering conference*. ACM, 2008, pp. 5–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1342215

[91] D. Canali et al., "A quantitative study of accuracy in system call-based malware detection," in *Proc. of ISSTA*, 2012, pp. 122–132.

[92] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *Proc. of NDSS*, 2009, pp. 8–11.

[93] X. Wang and R. Karri, "NumChecker: Detecting kernel control-flow modifying rootkits by using Hardware Performance Counters," in *Proc. of DAC*, 2013, pp. 1–7.

[94] R. Casula, "Shellshock security vulnerability," 2014.

[95] Virusshare, May 2014. [Online]. Available: http://virusshare.com/

[96] VX Heaven, Jan. 2014. [Online]. Available: http://vxheaven.org/

[97] Virustotal-free online virus, malware and url scanner. [Online]. Available: https://www.virustotal.com

[98] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[99] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. of ACSAC*. IEEE, 2007, pp. 421–430.

[100] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proc. of CCS*, 2010, pp. 399–412.

[101] J. Demme et al., "On the Feasibility of Online Malware Detection with Performance Counters," in *Proc. of ISCA*, 2013, pp. 559–570.

[102] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised Anomaly-Based Malware Detection Using Hardware Features," in *Proc. of RAID*, 2014, pp. 109–129.

[103] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-Aware Processors: A Framework for Efficient Online Malware Detection," in *Proc. of HPCA*, 2015.

[104] M. Bahador, M. Abadi, and A. Tajoddin, "HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *Proc. of ICCKE*, 2014, pp. 703–708.

[105] M. Rahmatian, H. Kooti, I. Harris, and E. Bozorgzadeh, "Hardware-Assisted Detection of Malicious Software in Embedded Systems," *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 94–97, Dec. 2012.

[106] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Architectural enhancements for secure embedded processing," *NATO Security Through Science Series D-Information and Communication Security*, vol. 2, p. 18, 2006.

[107] M. Fiskiran and R. Lee, "Runtime execution monitoring (REM) to detect and prevent malicious code execution," in *Proc. of ICCD*, 2004, pp. 452–457.

[108] A. Kanuparthi, R. Karri, G. Ormazabal, and S. Addepalli, "A high-performance, low-overhead microarchitecture for secure program execution," 2012, pp. 102–107.

[109] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. of ASPLOS*, 2000, pp. 168–177.

[110] S. Andersen and V. Abella. (2004) Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. [Online]. Available: http://technet.microsoft.com/en-us/library/bb457155.aspx

[111] PaX Team (2003). PaX Non-Executable Pages Design & Implementation. [Online]. Available: http://pax.grsecurity.net/docs/noexec.txt

[112] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing." in *Proc. of USENIX Security*, 2013, pp. 447–462.

[113] S. Das, W. Zhang, and Y. Liu, "Reconfigurable dynamic trusted platform module for control flow checking," in *Proc. of ISVLSI*, 2014, pp. 166–171.

[114] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation," in *Proc. of DAC*, 2014, pp. 1–6.

[115] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli, "A survey of microarchitecture support for embedded processor security," in *Proc. of ISVLSI*, 2012, pp. 368–373.

[116] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing mlp-bp on fpgas: A study," *Neural Networks, IEEE Transactions on*, vol. 18, no. 1, pp. 240–252, 2007.

[117] M. Rahmatian et al., "Hardware-assisted detection of malicious software in embedded systems," *Embedded Sys. Letters, IEEE*, vol. 4, no. 4, pp. 94–97, 2012.

[118] L. Bossuet, G. Gogniat, and W. Burleson, "Dynamically configurable security for sram fpga bitstreams," *International Journal of Embedded Systems*, vol. 2, no. 1, pp. 73–85, 2006.

[119] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2008. [Online]. Available: http://doi.acm.org.ezlibproxy1.ntu.edu.sg/10.1145/2089125.2089126

[120] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

[121] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," *Information and Computation*, vol. 103, no. 2, pp. 299–347, 1993.

[122] S. Das et al., "Semantics-based online malware detection: Towards efficient real-time protection against malware," *Information Forensics and Security, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[123] C. Li, A. Raghunathan, and N. Jha, "Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system," in *Proc. of Healthcom*, 2011, pp. 150–156.

[124] D. Halperin et al., "Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses," in *Proc. of Security and Privacy*, 2008, pp. 129–142.

[125] N. R. Saxena and E. J. McCluskey, "Control-flow checking using watchdog assists and extended-precision checksums," *Computers, IEEE Transactions on*, vol. 39, no. 4, pp. 554–559, 1990.

[126] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in *Usenix Security*, vol. 13, 2013.

[127] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *Computers, IEEE Transactions on*, vol. 100, no. 3, pp. 264–276, 1987.

[128] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: runtime intrusion prevention evaluator," in *Proc. of ACSAC*, 2011, pp. 41–50.

[129] J. Salwan. Ropgadget - gadgets finder and auto-roper, accessed Dec. 2014. [Online]. Available: http://shell-storm.org/project/ROPgadget

[130] L. Le, "Payload already inside: datafire-use for rop exploits," *In Black Hat USA*, 2010.

[131] T. cker Chiueh and F.-H. Hsu, "RAD: a compile-time solution to buffer overflow attacks," in *Proc. of ICDSC*, 2001, pp. 409–417.

[132] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "Return-Less" kernels," in *Proc. of EuroSys*, 2010, pp. 195–208.

[133] A. K. Kanuparthi, M. Zahran, and R. Karri, "Architecture support for dynamic integrity checking," *Information Forensics and Security, IEEE Transactions on*, vol. 7, no. 1, pp. 321–332, 2012.

[134] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from Code Reuse Attacks," in *Proc. of HPCA*, 2013, pp. 258–269.

[135] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing." in *Proc. of USENIX Security*, 2013, pp. 447–462.

[136] D. Song et al., "Bitblaze: A new approach to computer security via binary analysis," in *Information systems security*. Springer, 2008, pp. 1–25.

[137] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *Computers, IEEE Transactions on*, vol. 100, no. 3, pp. 264–276, 1987.

[138] S. Liakh, M. Grace, and X. Jiang, "Analyzing and improving linux kernel memory protection: a model checking approach," in *Proc. of ACSAC*, 2010, pp. 271–280.

[139] J. Oakley and S. Bratus, "Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code." in *Proc. of WOOT*, 2011, pp. 91–102.

[140] I. Skochinsky, "Compiler internals: Exceptions and rtti," 2012. [Online]. Available: http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf

[141] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A simulation framework for CPU-GPU computing," in *Proc. of PACT*, 2012, pp. 335–344.

[142] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[143] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. of WWC-4*, 2001, pp. 3–14.

[144] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "Biobench: A benchmark suite of bioinformatics applications," in *Proc. of ISPASS*, 2005, pp. 2–9.

[145] J. McCalpin. Stream benchmark, accessed Oct. 2014. [Online]. Available: http://www.cs.virginia.edu/stream/ref.html

[146] Multi2Sim: A heterogeneous system simulator, accessed Oct. 2014. [Online]. Available: http://www.multi2sim.org

[147] Valgrind exp-bbv Basic Block Vector Generation Tool, accessed Oct. 2014. [Online]. Available: http://valgrind.org/docs/manual/bbv-manual.html

[148] Processor Specification, accessed Oct. 2014. [Online]. Available: http://ark.intel.com/products/28024

[149] T. Rains, M. Miller, and D. Weston, "Exploitation Trends: From Potential Risk to Actual Risk." RSA Conference, 2015.

[150] "Security updates for adobe flash player," http://helpx.adobe.com/security/products/flash-player/apsb14-07.html.

[151] "Vrt: Anatomy of an exploit: Cve 2014-1776," http://vrt-blog.snort.org/2014/05/anatomy-of-exploit-cve-2014-1776.html.

[152] "Here's that fbi firefox exploit for you (cve-2013-1690)," https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690.

[153] "Skylined. internet explorer iframe srcname parameter bof remote compromise." http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory/iframe.html.php,2004.

[154] "Microsoft corporation. microsoft security advisory (961051)," http://www.microsoft.com/technet/security/advisory/961051.mspx,Dec.2008.

[155] "Multi-state information sharing and analysis center. vulnerability in adobe reader and adobe acrobat could allow remote code execution," http://www.msisac.org/advisories/2009/2009-008.cfm,Feb.2009.

[156] A. van de Ven, "New security enhancements in red hat enterprise linux v. 3, update 3," *Raleigh, North Carolina, USA: Red Hat*, 2004.

[157] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 417–432.

[158] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM workshop on Scalable trusted computing*. ACM, 2009, pp. 49–54.

[159] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller, "Detecting code reuse attacks with a model of conformant program execution," in *Engineering Secure Software and Systems*. Springer, 2014, pp. 1–18.

[160] "Transparent rop detection using cpu performance counters," THREADS Conference 2014 https://www.trailofbits.com/threads/2014/transparent_rop_detection_using_cpu_perfcounters.pdf.

[161] S. Cristalli, M. Pagnozzi, M. Graziano, A. Lanzi, and D. Balzarotti, "Micro-virtualization memory tracing to detect and prevent spraying attacks," in *USENIX Sec. Symp.*, 2016, pp. 431–446.

[162] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn, "Nozzle: A defense against heap-spraying code injection attacks." in *USENIX Sec. Symp.*, 2009, pp. 169–186.

[163] X. Chen, A. Slowinska, and H. Bos, "On the detection of custom memory allocators in c binaries," *Empirical Software Engineering*, vol. 21, no. 3, pp. 753–777, 2016.

[164] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[165] L. Yuan, W. Xing, H. Chen, and B. Zang, "Security breaches as pmu deviation: detecting and identifying security attacks using performance counters," in *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM, 2011, p. 6.

[166] G. Wicherski, "Taming rop on sandy bridge. syscan (2013)."

[167] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Proc. of RAID*, 2014, pp. 109–129.

[168] "The enhanced mitigation experience toolkit," https://support.microsoft.com/en-us/kb/2458544.

[169] Z. Liu, "Advanced heap manipulation in windows 8." [Online]. Available: https://media.blackhat.com/eu-13/briefings/Liu/bh-eu-13-liu-advanced-heap-WP.pdf

[170] "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3," Tech. Rep., Jun 2016. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf

[171] B. Biggio, "Machine learning under attack: Vulnerability exploitation and security measures," in *Proc. of IHMMSEC*, 2016, pp. 1–2.

[172] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[173] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Arch. News*, vol. 41, no. 3, pp. 559–570, 2013.

[174] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proc. of STC*, 2011, pp. 71–76.

[175] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proc. of SP*, 2013, pp. 559–573.

[176] "Branch Prediction Accuracy," http://www.realworldtech.com/cpu-perf-analysis/5/.

[177] "Metasploit framework," http://www.metasploit.com/.

[178] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[179] C. Pierce, M. Spisak, and K. Fitch, "Capturing 0day exploits with perfectly placed hardware traps." [Online]. Available: https://www.blackhat.com/docs/us-16/materials/us-16-Pierce-Capturing-0days-With-PERFectly-Placed-Hardware-Traps-wp.pdf

[180] "Alexa," http://www.alexa.com/.

[181] B. Vlaszaty and H. Rohani, "Test the effectiveness of the enhanced mitigation experience toolkit using well-known attacks on well-known binaries," 2014.

[182] X. Li and N. Carlini, "iROP: Interesting ROP Gadgets," in *SOURCE Boston 2015*.

[183] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *Proc. of SP*, 2015, pp. 745–762.