UNIVERSITY OF CALIFORNIA,
IRVINE

Securing Computer Systems Through Cyber Attack Detection at the Hardware Level

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computing Engineering

by

Congmiao Li

Dissertation Committee:
Professor Jean-Luc Gaudiot, Chair
Professor Nader Bagherzadeh
Professor Tony Givargis

2020

DEDICATION

To

my family and friends

for their continuous support and encouragement

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my committee chair, Professor Jean-Luc Gaudiot for his guidance and support throughout my years at UCI. I would like to thank the committee members, Professor Nader Bagherzadeh and Professor Tony Givargis, for their valuable feedbacks to my research.

I would like to extend my gratitude to my colleagues and friends at PASCAL lab for their friendship, timely comments and suggestions during my work.

Finally, I would like to thank my father for his continuous encouragement during this journey. I am also very grateful to all my friends for their support.

CURRICULUM VITAE

Congmiao Li

EDUCATION

2005-2009	Bachelor's Degree (with Honors) in Computing (Computer Science), Minor in Mathematics, National University of Singapore
2007	Exchange student in IT Department, Uppsala University, Sweden
2010-2013	Master of Science in Electrical Engineering, National University of Singapore
2013-2020	Ph.D. in Computer Engineering, University of California, Irvine

FIELD OF STUDY

Computer Architecture and Cyber Security

PUBLICATIONS

C. Li and J.-L. Gaudiot. Challenges in detecting an "evasive spectre". IEEE Computer Architecture Letters, 2020.

C. Li and J.-L. Gaudiot. Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters. In Proceedings - 2018 30th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2018, pages 25–28, Feb 2019.

C. Li and J.-L. Gaudiot. Detecting malicious attacks exploiting hardware vulnerabilities using performance counters. In Proceedings - 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), volume 1, pages 588–597, Jul 2019.

ABSTRACT OF THE DISSERTATION

Securing Computer Systems Through Cyber Attack Detection at the Hardware Level

By

Congmiao Li

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2020

Professor Jean-Luc Gaudiot, Chair

Over the past decades, the major objectives of computer design have been to improve performance and to reduce cost, energy consumption, and size, while security has remained a secondary concern. Meanwhile, malicious attacks have rapidly grown as the number of Internet-connected devices, ranging from personal smart embedded systems to large cloud servers, have been increasing. Traditional antivirus software cannot keep up with the increasing incidence of these attacks, especially for exploits targeting hardware design vulnerabilities. In this research, we propose to add additional layer of malware detection mechanism at the hardware level to improve overall system security by monitoring anomalies in semantic (control flow) and sub-semantic (microarchitectural) behaviors.

We developed a real-time application-specific malware detection system which is implemented in tightly coupled FPGA to monitor the Control Flow Integrity (CFI) of running programs on CPU. It runs in parallel with the CPU being monitored and provides real-time feedback to the system in case of control flow violation. The experiment result shows that the solution is scalable for large applications in embedded systems.

The impact of malicious attacks targeting hardware vulnerabilities can be catastrophic and widespread and no software patch can completely fix the problem. We propose to detect such attacks by monitoring microarchitectural features deviations. This is done by collecting related data from existing hardware performance counters. We take *Rowhammer* (exploits DRAM disturbance error vulnerability) and *Spectre* (exploits speculative execution and side channel vulnerabilities) attacks to demonstrate the feasibility and effectiveness to detect such attacks using microarchitectural features. An online detection method is adopted to detect malicious behaviors during the attack at early stage rather than offline detection after the damage is done. The experimental results show promising detection accuracy. However, the attacker may attempt to evade detection by reshaping the microarchitectural profile of *Spectre* to mimic benign programs. Future malware detector needs could be evasion resilient by randomly switching between multiple detectors using different features and sampling periods.

Chapter 1. Introduction

The number of cyber attacks is growing rapidly while more transactions are transmitted through cyberspace. Security for these systems is difficult long-term and open problem. It also involves issues beyond those being addressed in current desktop systems. Computers are the basic machines behind all the cyber activities. However, security has not been seriously considered in the foundations of computer design. Traditional software-based solutions may not be effective for preventing and detecting certain new security threads, especially for embedded systems. This proposal plans to research on the possible ways to improve computer security from the hardware level and examine their effectiveness and cost.

A major portion of cyber-attacks is malware, it usually refers to any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system [1]. They can be categorized as viruses, Trojan horses, spywares and other intrusive code [2]. According to Symantec's analysis, there were nearly 1 million new malware threads per day in 2014 [3]. The sheer number and variety of known and unknown malware is part of the reason why detecting malware is a difficult problem.

Security protection strategies have to be flexible enough to adapt to large number of new attacks. Therefore, they have been implemented in software due to the flexibility of software. For example, antivirus (AV) software is the most common way to protect computers against malware. However, software-based solution may not be sufficient when new attacks increase at increasing rate, especially in the context of embedded system. Because AV is itself software, and its complexity makes it more prone to bugs. The underlying operating system and hypervisor may also have bugs. Attackers can exploit these bugs to disable the protection from AV. In

addition, AV relies on static analysis to detect malware, but attackers can write different code variants to escape from detection. Moreover, many embedded systems also have processing power, deadline and power consumption constraints. Traditional software-based solutions may not meet the requirements of protecting embedded systems.

Traditional malware detection techniques are envisioned to be implemented in software. They usually make use of the analyzed malware characteristics to detect abnormal behaviors. As malware signature schemes grow richer for more accurate detection, the detection system tends to have large performance overheads. Due to increasing number of cyber-attacks especially those attacks targeting hardware design vulnerabilities, it is important to improve security at the hardware level.

In this dissertation, we seek to firstly survey and explore ways to improve the computer system security using hardware-based solutions in the following two aspects. First, we identify some of the traditional detection techniques that can be implemented in the dedicated hardware to achieve higher performance and finer granularity as described in Chapter 3. These techniques should be carefully chosen so that they have small variance across different system and malware types to be easily implemented in hardware. Second, we also look for novel lightweight hardware-based techniques to detection malware especially those targeting hardware vulnerabilities as discussed in Chapter 4. We would like to experiment with novel designs that utilize techniques from different perspectives to achieve better security, lower overheads and more robustness.

Chapter 2. Literature Review

2.1 Classification of Malware Detection Techniques

Malware detectors implements malware detection technique(s) on the same system it is trying to protect or on a separate system. Malware detection techniques can be broadly categorized into signature-based (or misuse) detection and anomaly-based detection [4]. Signature-based detection uses the characterization of what is known to be malicious to decide whether a program is malware. On the other hand, anomaly-based detection uses its knowledge of what constitutes normal behavior to make the decision. This technique has training phase and detection phase. The normal system behavior profile is created during the training phase by different machine learning algorithms. Specification-based detection, as a special type of anomaly-based detection uses a manually predefined set of specifications of what is normal behavior to decide the maliciousness of a program. It usually has lower false positive rate than the usual anomaly-based detection.

Each detection technique can implement static, dynamic, or hybrid approach according to how the information is gathered to detect malware as shown in Figure 1. A static approach tries to detect malware before the program under inspection (PUI) executes using the information collected from static analysis such as syntax or structural properties of the program whereas a dynamic approach tries to detect malicious behavior during program execution using runtime information. Hybrid techniques try to combine the two approaches using both static and dynamic information.

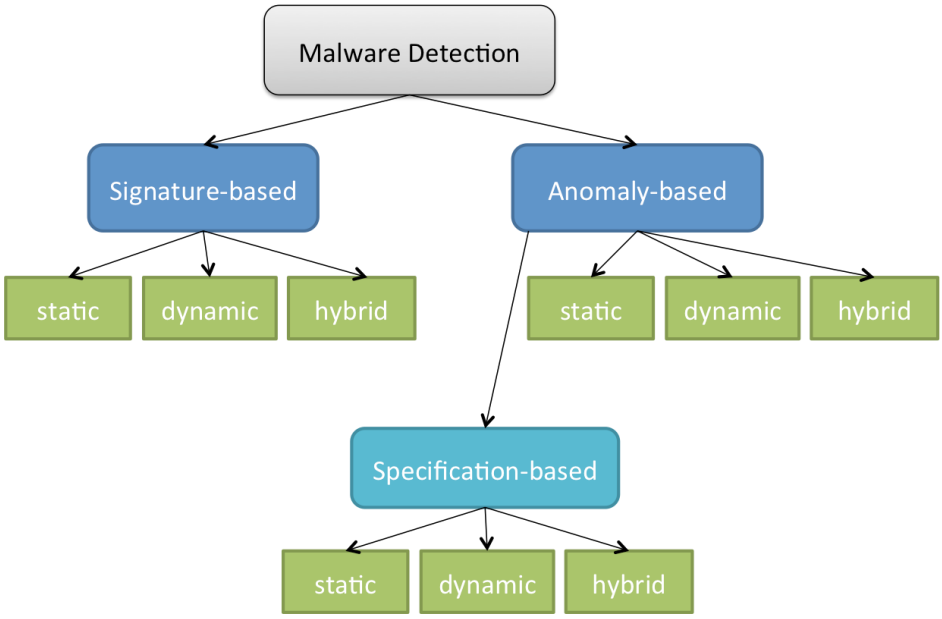

Figure 1. Classification of malware detection techniques.

Sung et al. [7] proposed a signature-based detection method named Static Analysis for Vicious Executables (SAVE). A given virus is characterized by a sequence of Windows API calls. If the Euclidean distance between the sequences of API calls of a known virus and the PUI is less than a certain threshold, then the PUI is identified as malicious. Other static signatures such as 3-tuple template of instruction, variables, and symbolic constants [8], assembly code [9]. Traditional pattern matching methods in static signature-based detection cannot detect self-encrypting and polymorphic viruses. Dynamic and hybrid signature-based approaches that also use malware run-time signatures were proposed. For example, Ilgun et al. [10] model a known attack pattern as a state transition diagram. Ellis et al. [11] use different behavioral signatures of worms such as data flows, server-to-client, alpha-in/alpha-out (worm sends similar data across nodes) signatures. Mori et al. [12] developed a tool to decrypt the virus payload in an OS emulator and then perform static analysis.

In anomaly-based detection, Wang and Stolfo [13] models the correct program behavior as expected payload in byte frequency distribution and compare incoming payloads with the model using Mhalanobis distance. Lee and Stolfo [14] created an rule set for various security critical aspects of the target host using data mining techniques. Others model correct behaviors as system call sequences [15], Finite State Automata (FSA) where state transactions are triggered by system calls [16], frequency of system calls [17]. There are also techniques which using static information such as file type and structure [18]. Wang et al. [19] proposed a hybrid anomaly-based detection technique to detect ghostware that alter the return values of certain system calls and hide its existence from OS utilities.

Specification-based detection defines certain rule set to model the normal program behavior. For example, Masri et al [20] verify applications at runtime according to the defined security policies, such as a policy can be defined to prevent illegal information flow from directory marked as "SenstiveSources". Ko et al. [21] defined a specification language to specify the intended behavior of privileged program. Linn et al. [22] proposed to add an interrupt address table that contains the system call number and the address after the system call in the program binary file to prevent unexpected system calls during runtime. Many static specification-based detection techniques were also proposed to analyze the maliciousness of a program before its execution through analyzing assembly code [23][24], annotations created by compiler [25]. Hybrid specification-based detection techniques usually extract static information before program execution, and monitor program behavior during runtime according the static specifications. For example, Wagner and Dean [26] proposed to derive a CFG (control flow graph) from source code analysis to represent the system call trace. During runtime, it sends an alarm if a system call was made that was not defined in the previously derived CFG.

In general, the main advantage of anomaly-based detection is its ability to detect zero-day attacks [5] which are previously unknown to the detector. However, it has high false alarm rate and is hard to decide what features should be learned during the training phase. Specification-based technique use pre-defined rules to reduce the false alarm rate of anomaly-based detection. Signature-based detection searches the repository of all known malicious behavior signatures to assess whether the PUI contains a known signature. The major drawback of this method is that it cannot detect zero-day attacks.

2.2 Hardware-assisted malware detection

Many malware detection techniques with software-based implementation experienced large computational complexity. It causes performance degradation, especially for embedded system with limited processing power. Moreover, such systems are not able to monitor program behavior at a very fine level of granularity that creates opportunities for attackers. In addition, software-based solution itself may contain bugs and vulnerabilities.

The basic concept of using a dedicated hardware unit or coprocessor to facilitate secure execution has roots dating back in 1997 [27] and [28]. They were developed to protect the crypto-processors that were used to store cryptographic keys and execute cryptographic algorithms. Hardware based malware detection systems have also been explored in different domains. Basically, they can be categorized as host-based and network-based solutions, where host-based systems monitor the program execution on individual host and network-based ones monitor the network traffic. Our work will focus on host-based solutions.

2.2.1 Malware Detection with CFI (Control-Flow Integrity) Enforcement

Many hardware-assisted detection techniques were evolved from previously proposed software-based solutions to achieve better performance and protection. In section 2.1, most of the reviewed techniques for host-based runtime malware detection use function calls or system calls to either model the malware behavior or the correct program behavior during runtime. They all try to enforce the basic safety property namely Control-Flow Integrity (CFI) [29], because most of the current attacks subvert machine-code execution and alter the program control-flow. Currently, CFI enforcement is still one of the most effective general approaches in preventing runtime attacks. As an anomaly-based (specification-based) detection method, it characterizes the correct behavior as permitted control flow as defined in CFG and checks program flow at runtime.

Arora et al. [30] used a dedicated hardware to build a hierarchical runtime framework. It is designed to monitor the program flow at basic block level and to check the integrity of instructions at runtime. Mao et al. [31] also proposed a monitoring system using the same approach but their finest granularity level can reduce to a single or few instructions. Both of the designs were simulated in SimpleScalar for performance study. Other CFI enforcement work with hardware implementation by Ragel et al. [32] were able to detect faults by checking return addresses and memory boundaries. Zhang et al. [33] check malicious behavior by verifying the program execution path dynamically, which requires significantly large amount of computational and storage resources.

In more recent research, CFI enforcement has shown the effectiveness in detecting malicious behavior in different platforms at hardware and software levels. The original proposal

of CFI experienced large performance overhead averaged at 21%. Thus, many researchers proposed coarse-grained CFI solutions with relaxed polices (e.gl, allowing returns to target any instruction following a call instruction). CFI approach has been implemented in smart phones powered by ARM processors. Davi. et al. [34] proposed a software-based framework (MoCFI) to perform CFI enforcement at runtime according to application binaries analysis on iOS devices. CFI enforcement was added during compilation process in an instrumentation layer of the compiled binaries for iOS device [35]. Microsoft has added a new security mechanism called Control Flow Guard (CFG) in Windows 10 and in Windows 8.1 update 3. They also built CFG compiler support in Visual Studio 2015 [39]. It focuses on mitigating problems if an indirect call is exploited and an invalid target is called instead. In x86 architecture, kBouncer [36] validates the Last Branch Recording (LBR) history table of recent Intel processors according to a coarse-grained CFI policy. CFIMon [37] detects control-flow violation using performance counters and Intel's Branch Trace Store (BTS) and only cause average 6.1% performance overhead for typical server applications. CFIMon relies on an offline training phase to collect a legal set of target addresses for each branch instruction.

There has been an arms race between the attackers and defenders. As we are trying to build more secure system and more powerful malware detector, on the other hand, attackers are developing more sophisticated attacks to overcome protection and compromise the system. Traditionally, code injection through stack or heap manipulation is one of the major attacks. But many processors have recently included hardware-enforced security features (e.g. no-execute (NX) in x86 architecture's paging scheme), which significantly prevented traditional code injection and user arbitrary code execution.

Attackers seek other ways to hijack the control flow and bypass the hardware protection by reusing the existing machine code ending with a ret instruction (called gadgets) without code injection. This early form of code reuse attack finds the useful gadgets to launch the attack by using the fact that these gadgets are located at known addresses in memory. Address-space layout randomization (ASLR) [40] was proposed to prevent against such attacks by randomizing the location of data and code region. Many general-purpose operating systems have implemented ASLR (e.g. Windows, Linux, OS X, Solaris, iOS, Andriod, …etc.). Later, different fine-grained ASLR techniques [41-46] were proposed to defend memory disclosures.

However, recent just-in-time (JIT) code reuse attack presented in [47] was able to dynamically discover gadgets on the fly even under the condition of fine-grained ASLR. It shows that randomizing the location of data and code is not a promising solution in the long run. This further corroborates that CFI enforcement is a more effective approach in preventing any kind of attacks that aim to redirect program logic. Recent papers [49, 50] have shown coarse-grained CFI policies can be undermined. Therefore, developing a more efficient fine-grained CFI enforcement mechanism is receiving increasing attention in preventing control flow hijacking.

To achieve fine-grained protection against code-reuse attacks, the authors of [38, 48] proposed Hardware-Assisted Flow Integrity eXtension (HAFIX) which adds CFI label related instructions in the instruction sets of Intel Siskiyou Peak and SPARC LEON3 to confine returns to active call sites. They provide hardware-assisted protection on the backward edges (returns) of CFG and assume the target system has software-based CFI protection for forward edges and implemented protection against code injection. This approach will potentially increase the

memory bandwidth and cache pressure. It also requires changes on the compiler to emit new CFI instructions.

2.2.2 Malware Detection with Performance Counters

Researchers have also explored other characteristics to distinguish malicious and correct program behaviors so that to detect malware efficiently besides monitoring the control flow. There have been research papers that use dynamic microarchitectural execution patterns from hardware performance counters to detect malware. Hardware performance counters are special-purpose registers that monitor hardware-related events such as cache misses or branch mispredictions. They are initially designed for performance analysis and fine-tuning. The registers are organized by the performance monitoring units. They require high-level privileges to be accessed, so attackers cannot directly use them for malicious attacks. Compared with higher-level features observed in OS and applications, microarchitectural features are simpler to audit and harder for attacker to exploit.

Demme et al. [51] examined the feasibility of building a hardware malware detector using dynamic performance data from existing performance counter. They adopted the signature-based detection approach that characterizes malware runtime behaviors with fine-grained microarchitectural trace features (L1 exclusive hits and arithmetic micro-Ops executed). A malware classifier was trained offline with labeled data collected from running recent Android malware on ARM processors and Linux rootkits on Intel platforms using standard machine learning algorithm such as KNN or Decision Trees. The result shows the data from performance counters can be used to identify malicious Android packages with nearly 90% accuracy and 3% false positives for some mobile malware using relatively simple classification algorithms.

Experiments with Linux rootkit detection shows less accurate detection results. This is because rootkits usually do not run independently, but dynamically interrupt program control flow. Thus, the collected training data doesn't record enough characteristics of rootkit behaviors. On the contrary, the detector worked perfectly well in detecting cache side-channel attack with 100% accuracy and no false positives. In addition, the authors also described different hardware implementation options to support online malware detection and methods to enable secure hardware updates of malware classifiers.

Compared with Demme et al.'s work on signature-based malware detector, Tang et al. [52] adopted the anomaly-based approach using both architectural and microarchitectural events to be able to detect zero-day attacks. Another major difference is that [52] detects malware at the exploitation stage (similar to CFI-based solutions), whereas [51] detects malware during the take-over stage when malicious payloads are running. They tried to model the normal execution behavior by monitoring the perturbations in the measurements from hardware performance counters (HPC) during exploitation using unsupervised learning algorithms. Due to the shorter execution duration of exploitation compared with payloads, the deviation is small but can be observed after amplification by power transform and temporal aggregation of multiple samples. They experimented with Internet Explorer 8 and Adobe PDF Reader 9 and the results show promising accuracy. However, for more sophisticated attacks the accuracy deteriorates. Therefore, this detector alone cannot provide enough protection, but it can be complementary to the previous signature-based detector to achieve better security.

Ozsoy et al. [53] proposed a detection framework called malware-aware processors (MAP) to reduce the false positives of detectors that rely only on hardware features collected from performance counter (also known as sub-semantic features). They adopted two-level

detection (TLD), which consists of a sub-semantic detector at the first level, and a more complex semantic detector at the second level. Subsequently, to improve the classification accuracy, Khasawneh et al. [54] proposed to use specialized detector for each specific type of malware and to use ensemble detectors, which combines the results from multiple specialized detectors. They also developed metrics to evaluate the performance advantages of TLD detection. Both of the two papers implemented the detection system in hardware by extending the AO486 open core for online detection.

Chapter 3. Reconfigurable Hardware Accelerated Malware Detection using Control Flow Integrity (CFI)

Security for computer systems has become an important issue as increasing number of devices are exposed to the cyber attacks especially for pervasive Internet of Things. We developed a real-time application-specific malware detection system that runs on a dedicated hardware implemented in FPGA. It runs in parallel with the system being monitored and provides real-time feedback to the running application in case of security violation. The system adopts the anomaly-based malware detection techniques which defines normal program execution behaviors. We model the correct behaviors as function call graph. The information is extracted from application source code offline before runtime. The system is able to detect any attack that causes the embedded application to deviate from the pre-defined permissible behavior with minimal hardware overheads.

3.1 Introduction

Computer systems are becoming increasingly pervasive in consumers' daily lives. From home appliances such as smart thermostats, locks, washing machines to large infrastructures such as smart power gird, urban traffic signage, more and more smart devices are being adopted to different application domains. Most of the devices require connection to the Internet to allow intelligent control. The number of connected devices is increasing dramatically. According to Cisco's report [55], there will be 50 billions of such embedded devices on the Internet by 2020. Security for these systems is a difficult long-term and open problem, because it involved issues beyond those being addressed in current desktop systems. Moreover, many embedded systems are developed by small teams, they cannot afford to provide security assurance.

The Internet connected embedded systems are vulnerable to cyber attacks. Among these systems there are large amount of critical control systems which may suffer from more severe damages due to cyber attacks. Since embedded system can causes physical changes to the environment and objects that is being controlled, the consequences of security breaches can result in significant societal disruption. For example, the Internet thermostat automatically adjusts the heating and cooling systems as measured environmental temperature changes to keep the comfort level in a house and achieve energy saving. Through Internet, the inhabitants can also update the desired temperature and other control parameters remotely before they arrive home. If an attacker gain access to the device, they can shut down the system or subject the house to extreme temperature. On the other hand, with users' persimmon, utility companies can get feedback and control all the connected thermostats to reduce peak loads. If the attackers launch a malware to the connected thermostats to turn on all the air conditioners at the same time, the sudden demand surge could cause power grid failure.

Although security issues for existing enterprise systems and embedded systems are both caused by Internet exposure of applications, embedded systems may have special constraints in term of timing, cost and power consumption to ensure security. For example, Enterprise applications usually are measured by each transaction. However, embedded systems usually operate periodic computations in each control loop with real-time deadlines. Sometimes, even a delay of one second can cause unstable control, which can have disastrous consequences. It is infeasible to have a system administrator to monitor the operation in real-time manually. A significant delay in detection of malicious intrusion to the system can also result in irreversible damages.

A major portion of cyber-attacks is malware, it usually refers to any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system [56]. They can be categorized as viruses, Trojan horses, spywares and other intrusive code [57]. According to PCWorld, there are average of 82,000 new malware threads per day in 2013 [58]. The sheer number and variety of known and unknown malware is part of the reason why detecting malware is a difficult problem.

In general, most of attacks compromise a system by executing malicious code at runtime. Attackers can achieve this through exploiting different system vulnerabilities and various channels. A common way is exploiting the lack of boundary checks in C/C++ programs to cause stack overflow. According to CVE vulnerability data which taken from National Vulnerability Database (NVD) [59], there are total 20,886 vulnerabilities from code execution, and 9,468 ones from overflow from year 1999 to 2014. They account for 38% of the total vulnerabilities through the 16 years. Therefore, we focus on detecting malicious code execution in this research.

Traditional software based malware detection systems usually analyze the system audit file to find out attacks when damage is already done. They are unable to detect attacks in real time and the time lag could cause disastrous consequence for embedded system as discussed earlier.

In this work, we propose a hardware based malware detection system to allow detection in near real-time. The system will run on a dedicated hardware that implemented with FPGA logic. It is directly integrated with embedded processor to monitor program execution and send feedback immediately to the processor. The main idea is to generate the function call graph for the program to be monitored through static analysis of its source code before execution. During

runtime, the processor sends execution progress to the detection system. If the execution violates the flow defined in the function call graph, then a malicious attack is reported.

3.2 Related Work

Wide range of software based malware detection techniques have been proposed by researchers previously. There are mainly two types of malware detection techniques, which are anomaly-based detection and signature-based detection [60]. Anomaly based detection defines the normal behaviors of a program and decide the execution is malicious if it violated the predefined rules or behaviors. In contrast, signature-based detection makes use the knowledge of known malicious attacks and inspects the program to determine if it has similar behavior as the known malware. Both of the approaches have their own strength and weakness under different situations. The main advantage of anomaly-based detection is it can detect previously unknown zero-day attacks.

Wagner et al. [61] proposed a malware detection approach by static analysis of program source code. It derives a control flow graph that models the normal system call trace. An alarm is triggered if it is found that during execution a system call was made that was not in the model. This type of model is referred to as the call graph model. As system call is a subset of function call, in our work, we consider all the function calls in general.

Many malware detection techniques with software-based implementation experienced large computational complexity. It causes large performance degradation for the embedded system with limited processing power. Moreover, such systems are not able to monitor program behavior at a very fine level of granularity. In addition, due to delay of detection, they may not be suitable for embedded system with real-time constraints.

The basic concept of using a dedicated hardware unit or coprocessor to facilitate secure execution has roots dating back in 1997 [62] and [63]. They were developed to protect the crypto-processors that were used to store cryptographic keys and execute cryptographic algorithms. Hardware based malware detection systems have also been explored in different domains. Basically, they can be categorized as host based or network based, where host based systems monitor the program execution on individual host and network based ones monitor the network traffic. Our work belongs to the host based solution.

Similar work to our solution has presented by Arora et al. [64] by using additional hardware to build a hierarchical runtime framework to monitor program flow at basic block level and check the integrity of instructions. Mao et al. [65] also proposed a monitoring system using the same approach but their finest granularity level can reduce to a single or few instructions. Our work reduced the computation complexity by analyzing the source code of program and monitoring only function calls.

Other related works by Abadi et al. [66] also use the information from control flow graph of a program to monitor attack. Ragel et al. [67] were able to detect faults by checking return addresses and memory boundaries. Zhang et al. [68] check malicious behavior by verifying the program execution path dynamically, which requires significantly large amount of computational and storage resources.

3.3 Proposed Malware Detection System

Our design of using dedicated hardware to detect malicious software in embedded CPU is aimed to achieve fine-grained, real-time detection with minimal performance and hardware overheads. It should have the following desired characteristics.

- The malware detection system is physically separate from the embedded processor. Even if the processor is compromised, the detection system could still be working properly.

- The end user and application programmer do not need to define security policies explicitly. The monitoring system is able to detect malware seamlessly by extracting normal behaviors of applications.

- With the assistance of dedicated hardware, runtime behaviors of applications can be detected at a fine granularity level compared with software-based solution. The hardware can also accelerate the malware detection process.

- As embedded systems usually have limited computation and storage resources, the solution should also try to minimize the overheads that introduced by implementing the detection systems.

To satisfy the design objectives, we use the invariant information extracted from the source code of application and implemented a dedicated hardware to check if actual program execution flow violates the predefined invariants.

The detailed hardware architecture of our malware detection system is shown in Figure 2. It includes two main subsystems, which are general purpose embedded system and real-time detection system. The two systems are running in parallel separately with feedback from each other.

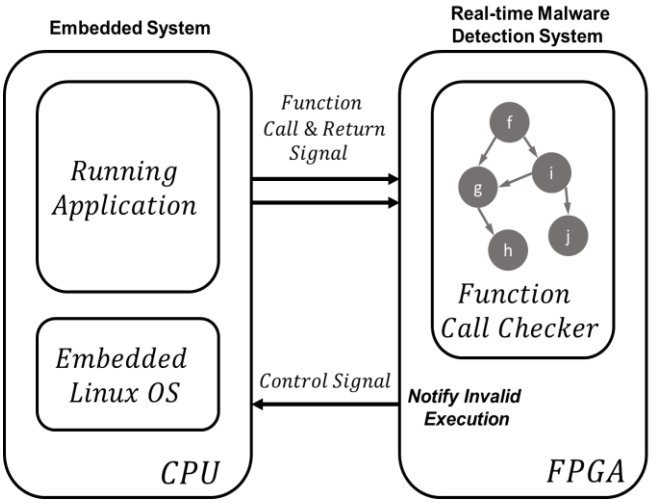

Figure 2. System architecture of malware detection system.

The general-purpose embedded system is the same as conventional one, which includes an embedded CPU, necessary I/O pins, memory and other elements to support application execution. The CPU is running embedded Linux operating system. User applications execute on top of the OS. The embedded system sends real-time function call and return signals of the target user application to be monitored to the detection system in real-time. This updates the detection system with current progress in term of function calls or returns of the running application.

The real-time detection system is implemented in FPGA. With the re-configurable feature of FPGA, the detection logic can be changed easily according to different running application in the CPU. The detection system consists of a function call checker for the running application. It checks the function calls or returns with the given information stream from the embedded system and pre-generated function call graph. The function call graph represents the sequence of permissible control flow. If the program flow deviates from the graph, a control signal will be

triggered to stop or notify invalid execution in the embedded processor. Any such invalid execution will be assumed as malicious attacks.

The function call graph is generated offline at compile time of the application. This process is considered as a static analysis of the source code without running the code. It firstly parses the source code to generate the syntax tree, and then find names of the defined functions and the functions they are calling to generate the call graph. The graph is then used to design the logic of FPGA for detection system. We assume that this process is carried out in a secured environment where hackers are not able to alter the source code or call graph. Before runtime, the application binary is loaded to CPU and the bit stream of detection system is loaded to the FPGA fabric.

We select function call flow as the application behavior to be monitored. Firstly, it can clearly distinguish a malicious execution flow from normal behavior. Secondly, it can be easily extracted from application source code automatically for any kind of programs. Finally, the complexity of the graph does not scale to an infeasible high level for large programs to ensure reasonable hardware overheads.

3.4 Implementation

The key component for the malware detection system is the function call checker. To implement it, we need to firstly generate the function call graph offline at a secure environment as mentioned previously. An example of call graph for an application that implements the Dijkstra's algorithm is shown in Figure 3. Each node in the graph represents a function, and an edge indicates a function call or return from one function to another.

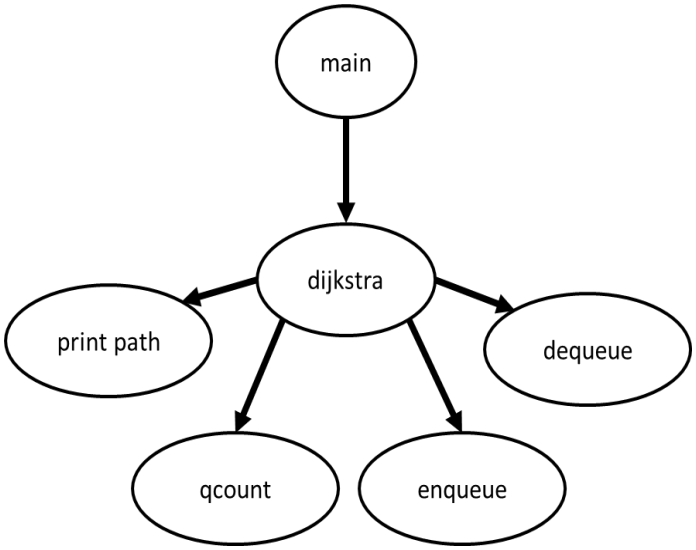

Figure 3. Function call graph (FCG) for Dijkstra's algorithm.

The function call graph (FCG) is then converted to a finite state machine (FSM) for hardware implementation. We use the Mealy type state machine to implement of the FSM where the outputs of the machine depend on both the inputs and current state. Compare with Moore type FSM used in [10] it may potentially have less states and transitions, so that the result may have less hardware overheads. In our design as shown in Figure 4, the FSM with N states takes the input $f = \{0, 1\}$, which is set to 1 if it is a function call and 0 represents a function return, and another set of input bits $I'\{i0, i1,\ldots,imax\}$ which is the binary representation of function index $i = \{0, 1,\ldots,N\text{-}1\}$, and also the value of current state index to decide the value of output. The FSM only have one-bit output v, which is set to 1 for valid transition and 0 for invalid one.

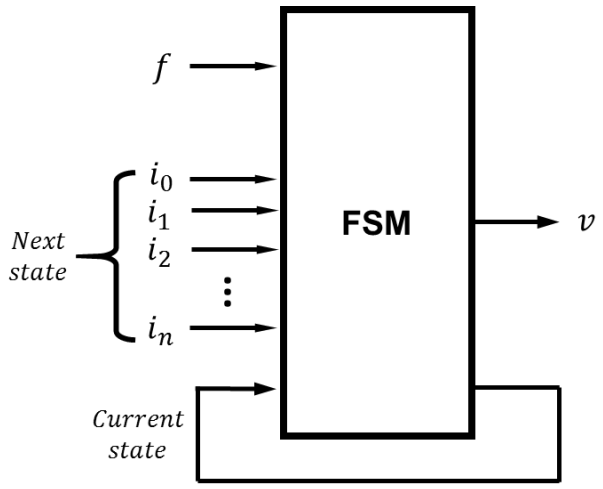

Figure 4. Hardware implementation block diagram for finite state machine (FSM).

A FCG with *N* functions is translated into a FSM with *N* states. Function with index *i* is mapped to state *i*. For each edge *eij* in FCG, add a transition from $state_i$ to $state_j$ with input $f = 1$ and $I'(imax…i0) = j$ and output $v = 1$ for function call, and another transition in the reverse direction from $state_j$ to $state_i$ with input $f = 0$ and $I'(imax…i0) = i$ and output $v = 1$ for function return. An example of converted FSM diagram for Dijkstra's algorithm is shown in Figure 5. Therefore, transitions with input values defined above will output 1 to indicate a valid execution, other input values or transition not defined will output 0 to report an invalid transition to the running application. The generated FSM is stored as DOT (graph description language) format for VHDL code generation in the next step.

After generating the finite state machine in DOT format, an automated tool was developed to interpret the DOT graph and translate it into VHDL file for hardware synthesis and implementation. We use the Vivado Design Suite from Xilinx to perform hardware synthesis and implementation for the Zynq-7000 All Programmable SoC. We chose this hardware because the Zynq-7000 consists a complete ARM®-based Processing System and a tightly integrated FPGA fabric which resemble the architecture of our proposed hardware-assisted malware detection

system. It also allows partial reconfiguration, which could be useful to extend our work in the future. Figure 6 shows the synthesized schematics for the sample program that implemented the Dijkstra's algorithm.

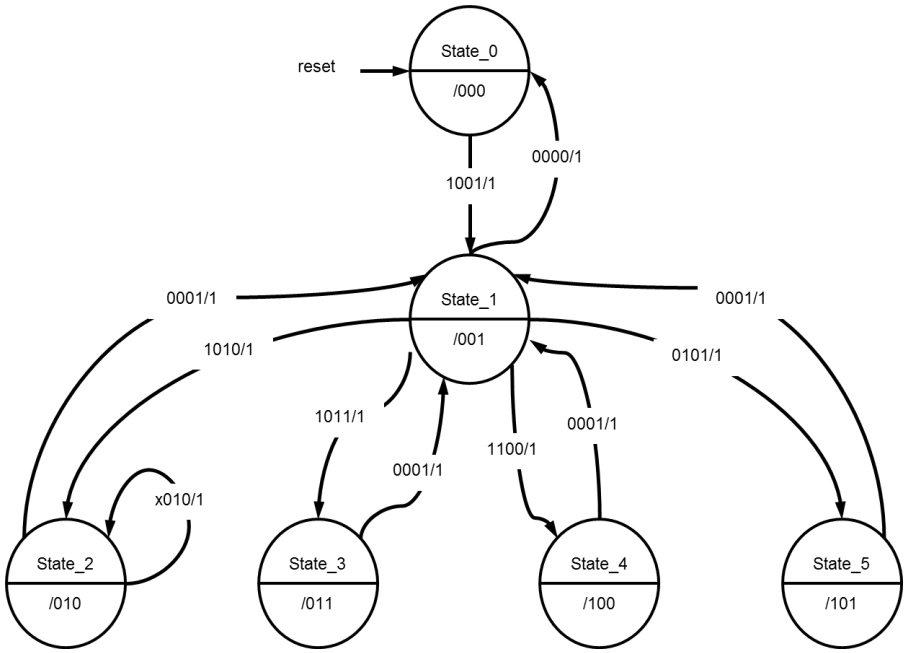

Figure 5. Function call graph (FCG) for Dijkstra's algorithm.

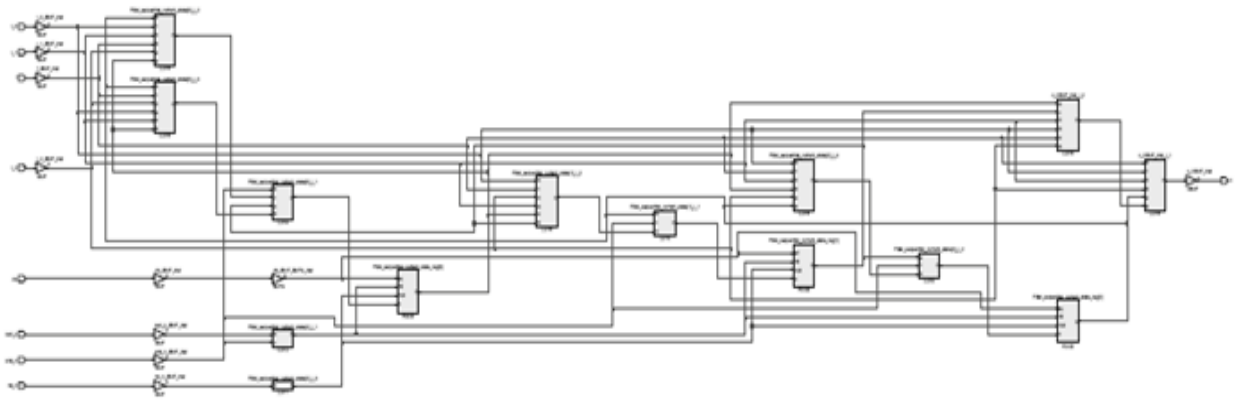

Figure 6. Synthesized schematics of FSM for Dijkstra's algorithm.

3.5 Attack Detection Evaluation

Our system modeled the intended program behavior and detects violation from the defined model. It is attack independent in the sense that it does not use information collected from past known attacks or the exact way by which they are perpetrated to help the detection. Therefore, this technique is not designed to protect particular type of vulnerabilities. It monitors the program behavior and anything that does not conform to the specified model is considered to be caused by a malicious attack.

The effectiveness of detection is determined by the modeled program behavior. We use the function call graph in our design. It is able to detect a violation due to function making a call to a disallowed callee or due to a function making a return to an invalid caller. This is effective to identify attacks that caused violations in function calls. It does not matter if the attacker altered the program flow through buffer overflow, gaining special privilege in a compromised system or other methods. However, as we only monitor the function call graph due to computational simplicity, it cannot detect violation at instruction level where attacker may insert a branch in the program binary.

3.6 Experimental Results

As our proposed system is application-specific and it scales with different application size, we chose the applications from MiBench benchmark suite [69] to generate realistic embedded system workloads. The suite is commercially representative embedded benchmark suite which contains applications from different domains including automotive/industrial, consumer, office, network, security and telecom. We chose representative applications within

different size range and ignore those trivial ones with less than five function calls to study the area overheads.

As discussed in section 3.3 we synthesized and implemented the FSM for function call checker on the Zynq-7000 from Xilinx which includes embedded arm processor as well as field-programmable gate arrays (FPGAs). The area overheads for different benchmark applications are listed in Table 1.

Table 1. Area Overheads for FSM.

Application	#Fns (states)	#Fn calls (transitions)	FF	LUT	LUT utilization %	I/O	BUFG
Dijkstra	6	6	3	11	0.02%	9	1
Patricia	7	9	3	12	0.02%	9	1
SHA	8	9	3	15	0.03%	10	1
Blowfish	12	20	4	34	0.06%	10	1
TypeSet	26	43	5	122	0.23%	11	1
MAD	49	65	35	249	1%	12	1
Lame	173	201	131	1700	3.2%	14	1
PGP	295	883	9	5261	9.89%	15	1
JPEG	435	649	9	4358	8.19%	15	1

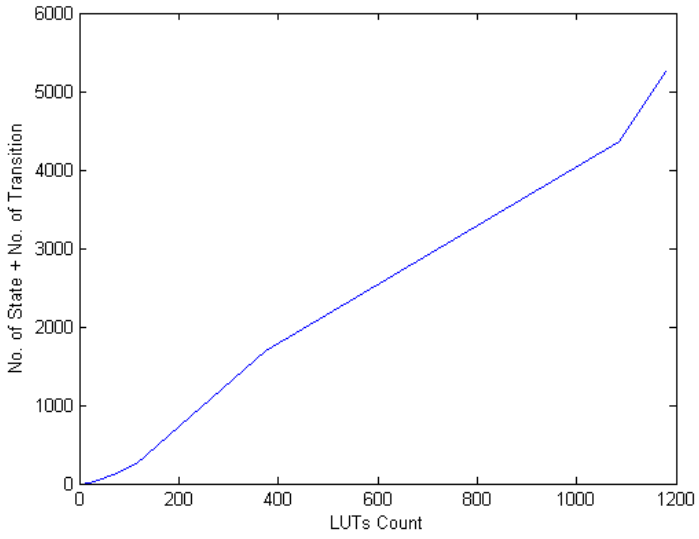

Figure 7. Lookup table LUT count for FSM.

In general, the result shows that the lookup table LUT count increase with increasing number of function and function calls in applications as shown in Figure 7. For the sample experiment, the LUT counts shows a near linear increase with rising complexity of FSM. For the largest application PGP, it utilized 9.89% of the LUT resources, which is still within a reasonable range. For applications with average size, the utilization rate is around 1%, which means with a relatively small among of extra hardware logic, the embedded system can enjoy much better security assurance. This is especially important for Internet connected critical control systems.

Compared with the FSM area overhead in work [64] implemented in Virtex2P family, for FSM with similar number of states and transitions, our results have smaller LUT count and utilization percentage due to different number of LUT input bits in the Zynq-7000 family and available hardware resources. With technology development in the FPGA industry, FPGAs are able to provide significantly reduced power, increased speed and lower cost solutions. The relative overheads compared with the total available hardware resources will be greatly reduced when implementing hardware based malware detection system. This will also enable more sophisticated detection techniques to be implemented in hardware with evolving malware detection techniques.

3.7 Conclusion

In this section, we implemented a hardware-based malware detection system for embedded applications. The anomaly-based solution is able to detect attacks in near real time with hardware acceleration. It uses the function call graph from source code analysis to model the correct program behavior, so that to be able to detect any prohibited function calls or returns caused by different security threats. The Experiment result also shows that the solution is

scalable for large applications in embedded systems. As we only experiment on monitoring one application at a time, Future work could be carried out for monitoring multiple applications running applications at the same time and dynamically reconfiguration of the detection system for switching between different embedded applications at runtime.

Chapter 4. Detection of Attacks Targeting Hardware Vulnerabilities

Modern computers introduced cache between main memory and the CPU to overcome the processor-memory speed gap. The timing difference between cache hits and misses can be exploited by attacker to leak critical information such as cryptographic keys through software cache-based side-channel attacks. They can be the primitives for more advanced attacks like Meltdown and Spectre. Traditional antivirus software cannot keep up with the increasing incidence of these attacks, especially for exploits targeting hardware design vulnerabilities. For example, as DRAM process technology scales down, it becomes easier for DRAM cells to electrically interact with each other. For instance, in *Rowhammer* attacks, it is possible to corrupt data in nearby rows by reading the same row in DRAM. As *Rowhammer* exploits a computer hardware weakness, no software patch can completely fix the problem. Similarly, there is no software mitigation to the recently reported attacks *Meltdown* and *Spectre*. The attacks exploit microarchitectural design vulnerabilities to leak protected data through side channels. In general, completely fixing hardware-level vulnerabilities would require a redesign of the hardware which cannot be backported. In addition, traditional software-based antivirus can hardly detect such attacks, as they do not leave traces in system log files. In this chapter, we demonstrate that by monitoring deviations in microarchitectural events such as cache misses, branch mispredictions from existing CPU performance counters, hardware-level attacks such as *Rowhammer* and *Spectre* can be efficiently detected during runtime with promising accuracy and reasonable performance overhead using various machine learning classifiers.

4.1 Background

4.1.1 Side Channel Attacks

Confidentiality is an important aspect in ensuring information security. It is often interpreted as to protect information from unauthorized parties. Information can be leaked to unprivileged parties through unintended side channels. These unintentional side channels can be timing information, power consumption, electromagnetic radiation, light emission or sound. For side channel attacks on cryptosystem, by measuring and analyzing the difference from side channel information, the secret key could be recovered. Side channel attacks can be mounted quickly or implemented using readily available hardware with low cost.

Timing attacks are performed by measuring the time it takes for a system to perform operations. For example, by carefully measuring the time required to perform private key operations, an attacker can find fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems. Cryptosystems often take slightly different amount of time to process different input due to performance optimizations. Performance characteristics typically depend on both the encryption key and the input data. Timing measurements are fed into a statistical model that can provide the guessed key bit with some degree of certainty. Computing the variances is easy and provides a good way to identify correct exponent bit guesses. In many cases, the attack is computationally simple and often requires only known ciphertext.

4.1.2 Cache Side Channel Attacks

Modern processor architecture features such as shared caches can inadvertently enable side channel attacks. Low-level implementation detail of modern CPUS, namely the structure of memory caches, causes subtle indirect interaction between processes running on the same

processor. This leads to cross-process information leakage. In general, the cache forms a shared resource that all processes compete for, and it thus affects and is affected by every process. While the data stored in the cache is protected by virtual memory mechanisms, the metadata about the contents of the cache, and hence the memory access patterns of processes using that cache, is not fully protected.

In recent years, cloud-computing services provide virtualized system resources to end users, supporting each tenant in a separate virtual machine (VM). High resource utilization in the cloud is achieved by resource sharing, where cloud providers co-host multiple VMs on a single hardware platform, relying on the underlying virtual-machine monitor (VMM) to isolate VMs and manage system resources. While virtualization creates the illusion of strict isolation and exclusive resource access, the virtual resources map to the same shared physical resources in reality, for example the last-level cache (LLC). The sharing creates the potential interference between co-hosted VMs. A malicious VM may learn information on data processed by a victim VM and perform side-channel attacks on the cryptosystem of the victim VM.

In Simultaneous Multi-Threaded (SMT) processors, caches are shared. An attacker can run a receiver (or observer) process simultaneously with the victim process on the same processor. This enables observation of the victim process's cache usage at run time. Osvik et al. [90] showed that after only 800 writes to a Linux dm-crypt encrypted partition, they were able to recover the full AES key using a simple cache side channel attack. The attacker accesses an array of his own data repeatedly so that he occupies all cache lines. During the execution of the victim AES encryption process, if the victim accesses a cache line, the attacker's data will be evicted. The next time the attacker accesses his data corresponding to this cache line, he will experience a cache miss. By measuring the memory access time, the attacker can learn the victim's cache

access pattern, based on which he can determine which lookup table entry is accessed during a key-dependent table lookup of AES implementation. The attacker then can recover the secret AES key.

In non-SMT processors, cache-based side channel attacks are also possible. Bernstein's attack on AES [73] demonstrated reported successful extraction of complete AES key from a network server on another computer through cache timing attack. The victim is a software module that can perform AES encryption for a user. The module is a "black box" and the user is only able to choose the input to the AES software module and measure how long it takes to complete the encryption. For most software implementation of AES running on modern processor, the execution time of an encryption is input-dependent and can be exploited to recover the secret key.

The attack consists of three steps. Firstly, during the learning phase, the attacker generates a large number of random plaintexts, then sends the plaintexts to the remote encryption server using a known key to record the encryption time for each plaintext. Secondly, during the attack phase, the attacker repeats the same operation in the first step except that the target unknown key is used. Finally, the attacker uses the two sets of timing profiles together with a correlation algorithm to recover the unknown key.

4.1.3 Meltdown Attack

Meltdown [70] allows a user space rogue process to read any physical, kernel or other processes' mapped memory by leveraging out-of-order execution to bypass the normal privilege checks that isolate the malicious process from kernel and other processes and then leak the memory content through cache side channel attacks. Mitigation of *Meltdown* involves changes in

kernel code to further isolate kernel memory from user-mode processes, which has been released by OS vendors through software kernel patches.

4.1.4 Spectre Attack

However, there are still no efficient software patches for *Spectre* attack until now. As report in [71, 72], the attack has two variants: bounds check bypass and branch target injection. The first variant exploits conditional branch mispredictions. For example, the victim function in Listing 1 receives integer *x* from an untrusted source. The function does a bound check on *x* to prevent the process from reading unauthorized memory outside *array1* to ensure security. However, speculative execution can lead to out-of-bounds memory reads. Suppose the attacker makes several calls to *victim_function()* to train the branch predictor expect taking the branch by feeding it with valid values of *x*, then calls the same function with an out-of-bound *x* that points to a secret byte in the victim's memory.

```
void victim_function (size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}
```

Listing 1: Conditional Branch Example

The attack usually consists of three phases. In general, it starts with the setup phase where the attacker prepares the side channel to leak the victim's sensitive information, and other necessary pre-requisites such as to mis-train the branch predictor to take erroneous execution path, and to load target memory location into registers, etc. In the following phase, the attacker diverts confidential information from the victim's context to a microarchitectural side channel by

exploiting different hardware vulnerabilities such as out-of-order execution or speculative execution. Then during the final phase, the attacker gains access to the secret data through the prepared side channel in the previous stages.

In the setup phase of *Spectre* attacks, the adversary prepares for three conditions. Firstly, a malicious out-of-bound value of *x* is chosen such that *array1[x]* points to a secret byte *s* in the victim's memory. The cache is then configured so that *array1_size* and *array2* are not in the cache, but the secret *s* in the cache. Lastly, the branch predictor is trained to expect the condition $x < array1\_size$ to be true.

During the second phase, the victim function starts by comparing the malicious value of *x* with *array1_size*. Because *array1_size* is not cached, the mis-trained branch predictor assumes the condition to be true while waiting for the value of *array1_size* to be loaded from memory. The processor speculatively executes the instruction inside the *if* condition. Reading secret data *s* from *array1[x]* is fast because it is a cache hit. Then s is used to compute the address of *array2[s*512]* and the program quickly starts to read data from this address in memory. While waiting for the read to return, the value of *array1_size* is finally arrived. The processor then realized the branch predictor took the wrong execution path and then roll back the register states. However, the cache state of *array2* is already affected by the memory read from the specific address related to the secret *s* and is not reversed.

In the final phase of the attack, using cache side channel attack such as the method proposed in [73] can identify which cache line in *array2* was loaded by measuring the timing differences. Such information can then be used to infer the value of secret byte. The attack can be repeated multiple times to reveal any unauthorized information from the memory.

The second variant of *Spectre* targets the indirect jump target prediction. It allows program to jump to an address in a register, an address in a memory location or an address in the stack. Like the first variant, if the target address is not in cache, the attacker can train the branch predictor to jump to an erroneous location. If the speculative execution leaves measurable differences in cache or other side channels, the attacker can infer secret information from the victim's memory.

The two variants discussed above rely on the changes in the state of the cache from speculative execution. There could be potential other variants of the attack where speculative execution affects other microarchitectural features. For example, instruction timing difference can be exploited where the time to execute an instruction depends on its operands. In general, any observable changes from speculatively executed code can potentially leak confidential information.

Mitigating the effects of *Spectre* is difficult because there are many variations of possible attacks. To prevent conditional branch vulnerability, speculative execution needs to be stopped on all potentially sensitive execution paths. However, insertion of such blocking mechanisms in all conditional branches and their destinations by compiler would severely degrade performance. Software mitigation to indirect branch vulnerability is even more challenging as there is no architecturally-defined method to block it and indirect jumps vary across different processors.

Besides patching the systems, it is also important to detect the malicious attack proactively and stop it at the earliest possible stage. Despite different variants of the *Spectre* attack, they all involve training the branch predictor to take the wrong execution path, and then leak the confidential information through an observable microarchitectural side channel.

For the case of conditional branch example in Listing 1, the attacker calls the victim function multiple times that causes the condition to be true. Therefore, we speculate that the branch misprediction rate will be reduced during the attack. In addition, the secret data are leaked through cache side channel. The attacker needs to flush the cache constantly to make sure *array2* and *array1_size* are not cached, so the cache miss rate is likely to be increased. By monitoring the deviation of these two microarchitectural behaviors, we could possibly detect the attack.

To further validate our hypothesis, we setup experimental attacks based on the proof of concept code in [72] and collect microarchitectural traces from hardware performance counters in modern processors to analyze the results in the subsequent sections. The proposed detection method can be further extended to other variations of *Spectre* attack by monitoring additional microarchitectural features according to different side channels being exploited.

4.1.5 Rowhammer Attack

To increase the capacity and reduce energy consumption, DRAM chips are getting denser. However, smaller cells can hold a lower amount of charge which reduces its noise margin. Higher proximity of cells also introduces electromagnetic coupling effects. Therefore, every time a DRAM row is read from a memory bank, the memory cells in adjacent rows leak a small amount of charge. If this happens frequently within one refresh cycle, the affected cells can potentially leak enough charge that the stored bit value will flip, a phenomenon known as *Rowhammer* [74].

In singled-sided *Rowhammer* attack [74], only one side of the neighboring row to the victim row was rapidly accessed. The newer version namely double-sided *Rowhammer* [75]

hammers both sides of the neighboring rows to increase the possibility of bit flip. To successfully trigger bit flip from CPU, memory accesses must bypass CPU caches and reach the target row in DRAM as frequently as possible. This could generate a large number of cache misses during short time period. To avoid caches, attackers can use cache flushing instructions [75, 76, 77], eviction buffers [78, 79, 80], and non-temporal store instructions [81].

4.2 Proposed Online Detection Approach

Our proposed detection approach firstly collects microarchitectural features from performance counters in every sampling period. For *Rowhammer* and the variant of *Spectre* attacks discussed in the previous section using conditional branch, we choose to monitor 4 events related to cache access and branch prediction behaviors, namely cache references, cache misses, branch instructions retired and branch mispredictions. The data are collected in clean environment where the computer runs typical desktop applications like web browser, video player, and text editors, as well as in environment under attack while running the same desktop applications respectively. The data are then labeled to train the machine learning classifier to classify the input data in each time interval. In runtime, the output time series from the trained classifier is fed into the online detection mechanism to decide if the system is under attack. This section describes the machine learning algorithms we used to train the classifier and our proposed online detection approach in details.

4.2.1 Machine Learning Classifiers

Machine learning algorithms can be used to train classifiers that determines which class y a given data set x belongs to. In most cases, the relationship between x and y is described by a probability distribution $P(x,y)$. The optimal class membership decision is to choose the class

label y such that the posterior distribution $P(y|x)$ is maximized [82]. We use supervised learning [83] to train the attack detector with a set of pre-labeled examples. There are two phases in supervised learning:

- **Training Phase**: to build classification model with the training dataset.
- **Testing Phase**: to classify new unseen input dataset using the model built in the training phase.

For each type of attacks, we collect data in 10 independent runs and use the same number (1,200) of samples from both classes to avoid any bias. Then, we randomly divide the collected data into training (80%) and test (20%) data, then separate the training data into training (80%) and validation (20%) data.

We choose the following three different machine learning algorithms to build the classifier with increasing complexity of the model:

Logistic Regression (LR): LR is a simple linear classification algorithm. It attempts to separate multi-dimensional input data points by hyperplanes where points on one side of the plane belong to the "normal" class and points on the other side belong to the "malicious" class. In general, the programs are not linearly separable, so LR gives a probability between 0 and 1 for the likelihood of a program trace being malicious. This probability is then converted into a binary decision by comparing it with a pre-defined threshold. Compared with other non-linear models, LR has fewer parameters and requires less time to train.

Support Vector Machine (SVM): SVM finds the optimal separating boundaries between data sets by modeling and solving the classification problem as a constrained quadratic

optimization problem [84, 85]. The degrees of nonlinearity and flexibility can be adjusted by using different kernel functions such as polynomial kernel, radial basis function kernel (RBF), *etc*. The classification result is dichotomous where the membership function could be either 0 or 1 without probability distribution. SVM has received considerable research interest over the past years because its performance is comparable with other non-linear models for many classification problems and it is less complex than artificial neural networks.

Artificial Neural Networks (ANN) ANNs consist in networks of perceptrons (Multilayer Perceptron - MLP) that approximate a classification function for the training data. ANNs usually contain an input layer, an output layer, and multiple hidden layers of perceptrons in between. It is a popular and promising machine learning technique due to its capability of mapping highly nonlinear data samples unlike any other statistical regression models. When there is no hidden layer, the network is actually identical to the LR model if the logistic activation function is used [28, 29]. By introducing nonlinear hidden neurons to the network, the output of the network can become a nonlinear function of the inputs. In classification problems, this can model the problem of nonlinear decision boundaries. Researchers have developed many models based on Back Propagation Networks (BPN) and Radial Basis Function Networks (RBFN) for highly nonlinear time series predictions [86, 87]. In general, ANN is more flexible than LR to model more complex data. However, it requires much longer time and more data to train the model.

4.2.2 Online Attack Detection

We propose to use online classification methods to detect malicious behaviors at runtime. To this end, we collect microarchitectural features periodically (every 100 ms). The

multidimensional data are then feed to a machine learning classifier to make decisions as to whether malicious code is being executed or not. The problem of detecting malicious attacks in real time is to make decisions according to the binary time series generated by the base classifier.

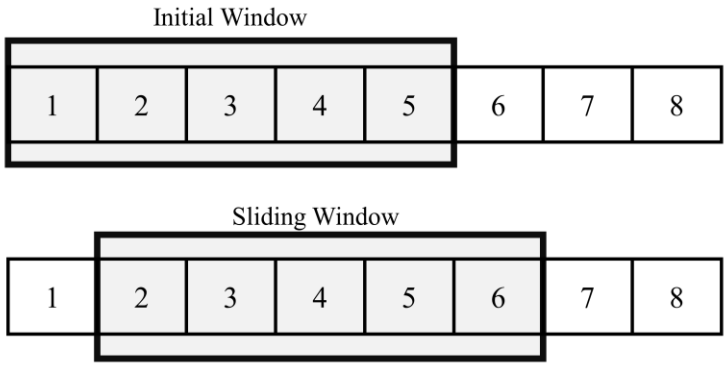

Figure 8. Example of sliding window.

To smooth the fluctuated time series data, a Weighted Moving Average (WMA) is used to filter out noise for better decision making by assigning a weight factor to each element in the time series. More recent data are assigned with higher weights. Then we segment the data using a sliding window [88, 89] to calculate the average of consecutive decisions within the current window. If the average is above a certain threshold, we conclude that an attack (malicious code) is in progress. Figure 8 illustrates an example of a sliding window process with a window size of 5. Each numbered segment corresponds to the classification result of each sampling period. The initial window contains the first 5 decisions. The data within this window are used to determine the final classification result for the current time. The detection runs continuously for the next period where the window slides to the right by one segment to cover data from segment 2 to 6, and it moves on to the next window accordingly.

In our experiments, we choose the window size to be 10 and sampling period of performance counters data collection to be 100 *ms*, which means our detector makes decision on whether the system is under attack every second. While the parameters could be tuned for different hardware systems, we selected the above numbers so that to get satisfiable detection accuracy without a major slowdown of the system.

4.3 Evaluation of Detection Performance

A key criterion to evaluate the detection performance is the accuracy of the model to make decisions on previously unseen data. To characterize the accuracy, we use metrics known as False Positives, False Negatives, True Positives (or Sensitivity) and True Negatives (or Specificity) defined as follows:

- **False Negatives (FN):** The percentage of positive instances incorrectly classified as negatives (percentage of misclassified malicious instances).
- **False Positives (FP):** The percentage of negative instances incorrectly classified as positives (percentage of misclassified normal programs).
- **True Positives (TP = 100% - FN):** The percentage of positive instances correctly classified as positives (percentage of correctly classified malicious instance).
- **True Negatives (TN = 100% - FP):** The percentage of correctly classified negative instances (The percentage of correctly classified normal instances).

A good detection approach should yield low false positives and high true positives (or low false negatives). To visualize the tradeoff between percentage of correctly identified

malicious instances and the percentage of normal instances misclassified, we use Receiver Operating Characteristic (ROC) graphs plotting TP against FP. In addition, to compare the performance of different models, we compute and compare the area under the ROC curve for each model. The Area Under Curve (AUC) score, also known as the c-index, provides a quantitative metric of how well an attack detection approach can distinguish between malicious and normal execution with a higher AUC value for better performance.

4.4 Experimental Setup

To examine whether performance counter data can be used to effectively detect attacks targeting hardware design vulnerabilities, we take *Rowhammer* and *Spectre* attacks as examples and collect events which have the potential to be affected by the attacks (this includes cache references, cache misses, branch instructions retired and branch mispredictions from running the attacks on top of normal programs and running typical benign applications alone respectively). The results are then analyzed and preprocessed before being used to train different classifiers to detect malicious behavior. In this section, we describe the details of the data collection mechanism and the system settings under attack and in normal conditions.

4.4.1 Data Collection Mechanism

We run the attack on a typical personal laptop with Debian Linux 4.8.5 OS on Intel Core i3-3217U 1.8 GHz processor with 3MB cache and 4GB of DDR3 memory from Micron. The Intel processor contains a model-specific performance counter monitor (PCM) and can be configured to count four different hardware events at the same time. According to the discussion on the nature of *Rowhammer* and *Spectre* attacks in Section 4.1, we choose the following available events for our system:

- Last-level cache reference event (LLC references)
- Last-level cache misses event (LLC misses)
- Branch instruction retired event (branches)
- Branch mispredict retired event (branch mispredictions)

We use the standard profiling infrastructure of Linux, *perf* tools, to obtain system-wide performance counter data. We run *perf* every 100ms to record the required data without excessively degrading the performance.

4.4.2 Test Environment Setup

In a clean environment, we sought to create realistic scenarios by randomly browsing popular websites (according to Wikipedia in FireFox) in different orders and by streaming videos from browser plug-ins. In addition, we also ran text editors to read and edit files. For data collection when the system is under malicious attack, we launched malicious attacks on top of normal running applications. To demonstrate the effectiveness of detecting attacks exploiting hardware vulnerabilities, we chose the double-sided *Rowhammer* attack and the *Spectre* proof of concept attack and ran them independently. The system status is reset after each run to ensure the measurements are independent across different clean and exploit runs. We collect overall performance counter data across the system rather than for individual processes. This is to make the classification problem closer to real world conditions, albeit more difficult.

4.5 Results

In the experiment described in section 4.2, we collected data from four performance counters at the same time periodically in 10 separate runs. Each run produced 1,200 malicious and normal samples respectively. In this section, we first analyzed the collected raw data to assess whether it is feasible to differentiate measurements in a clean environment from those under attack by visualizing the distribution of data. Then, we used different machine learning algorithms to train the classifier and build the real-time attack detector using the sliding window approach discussed previously.

4.5.1 Data Distribution Analysis

Our data collection mechanism produces 4-dimensional time series data. Each sample contains event counts for branch mispredictions, LLC misses, branches, and LLC references during the sampling period. We also calculate the branch miss rate (1) and LLC miss rate (2) for each interval as:

$$branch\ miss\ rate = branch\ mispredictions / branches \quad (1)$$

$$LLC\ miss\ rate = LLC\ misses / LLC\ references \quad (2)$$

We use *boxplot* to visualize the range and variance of the measured data for each individual microarchitectural feature. Figure 9 and Figure 10 give a direct indication of the feasibility to detect malicious attacks using one particular feature.

For *Rowhammer* attacks as shown in Figure 9, the LLC misses, LLC references and LLC miss rate are all concentrated in regions higher during the attack than during normal operations. The reason is that in order to successfully flip bits in the memory, the attacker has to bypass the

cache and access the neighboring rows next to the victim row in DRAM in rapid sequence. This results in more cache misses within a short time of period. Unlike previous work which only considers the impact on cache behavior, we also notice that the number of branch miss rate is greatly reduced. This is because the exploit keeps looping through the same instructions to repeatedly access the same rows in memory. This extra feature could help to reduce the error rate in the detection of *Rowhammer*.

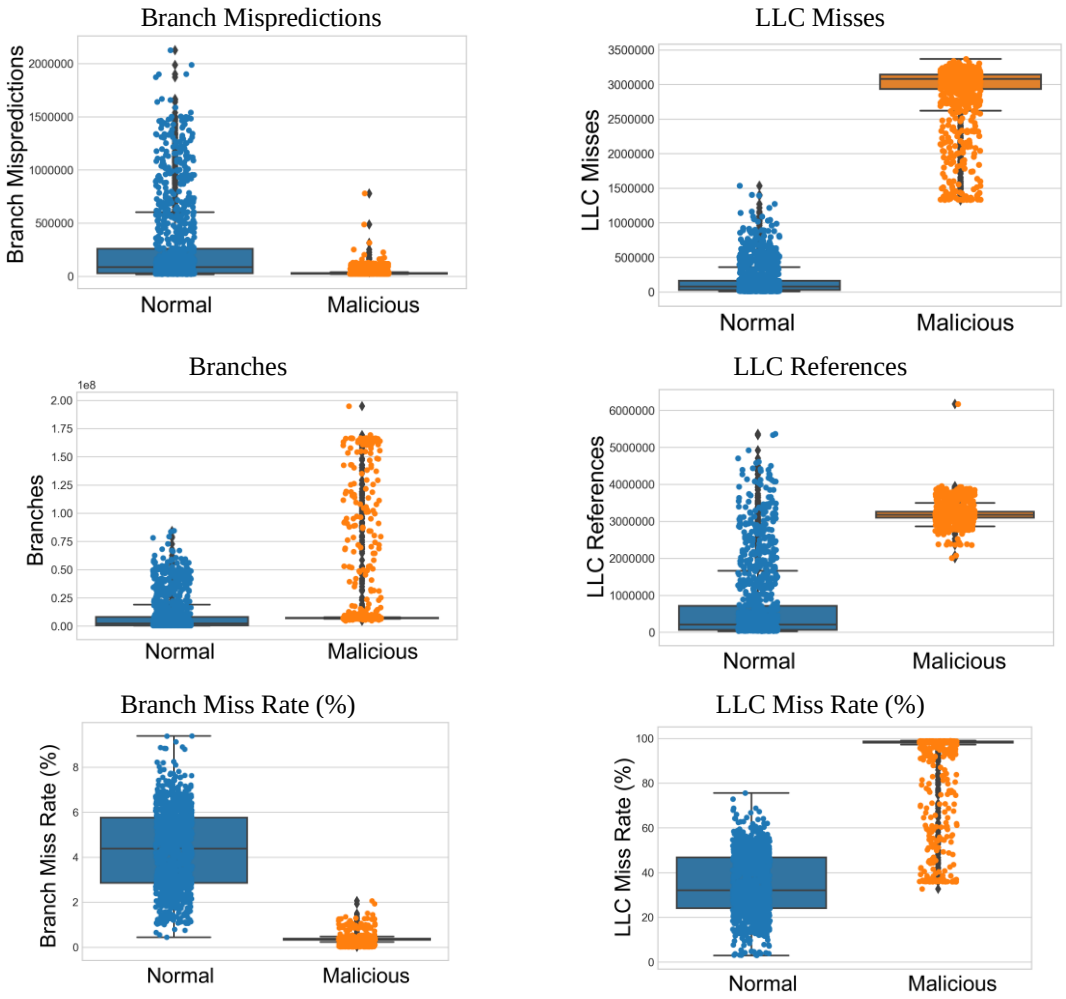

Figure 9. Distribution of microarchitectual features from performance counters for Rowhammer.

For the *Spectre* attacks shown in Figure 10, we observe an increased number of branches and branch mispredictions during the attacks. In contrast, the branch miss rate is decreased. This is because the attacker tries to train the branch predictor by calling the conditional branch many times with different input values that make the condition true. For the LLC, the number of references and misses are both increased with the miss rate concentrated on the higher percentage region due to cache side channel attacks. The experimental results validate our hypothesis proposed in section 4.1.4.

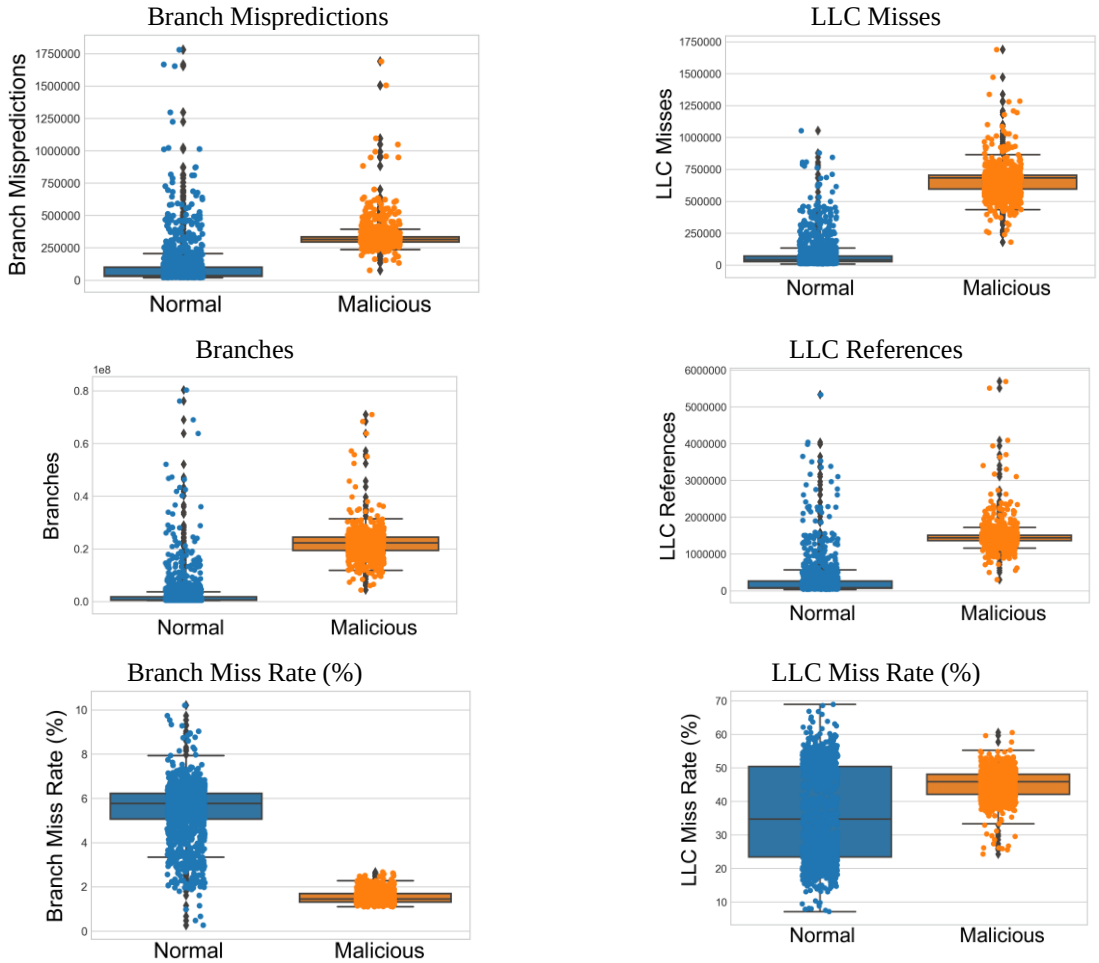

Figure 10. Distribution of microarchitectual features from performance counters for Spectre.

In addition, we also analyze the feasibility of distinguishing the collected performance counter data using more than one feature by plotting the sample points in 2D and 3D graphs with each dimension corresponding to one feature.

Figure 11 shows the distribution of normal and malicious sample points for *Rowhammer* attacks using only cache related features including LLC references and LLC misses. Figure 12 uses LLC references, LLC misses and branch miss rate. We can observe the data points of two different classes distribute in two different regions and the boundaries between the two are obvious in both plots. Therefore, we believe it is feasible to use the chosen microarchitectural features to detect *Rowhammer* attacks. In particular, with an additional branch feature, the boundary is clearer in Figure 5 which can lead to better classification performance.

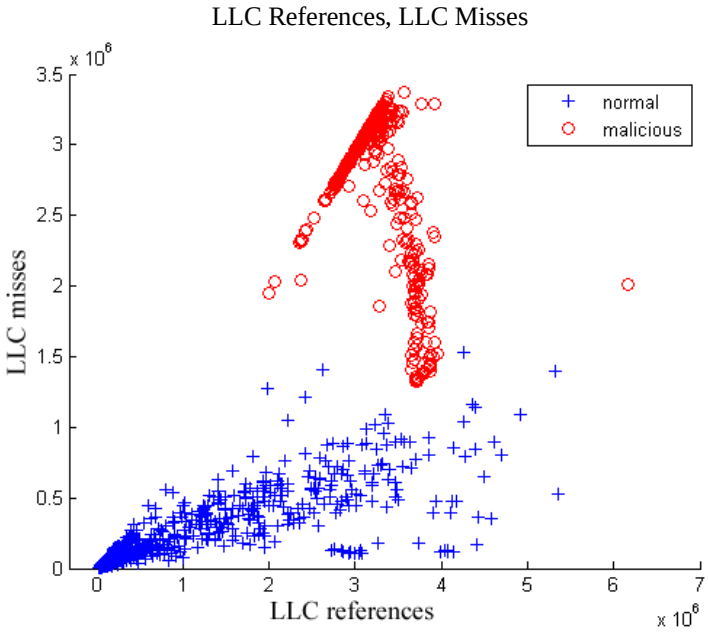

Figure 11. Distribution of branch miss rate and LLC miss rate features for Rowhammer.

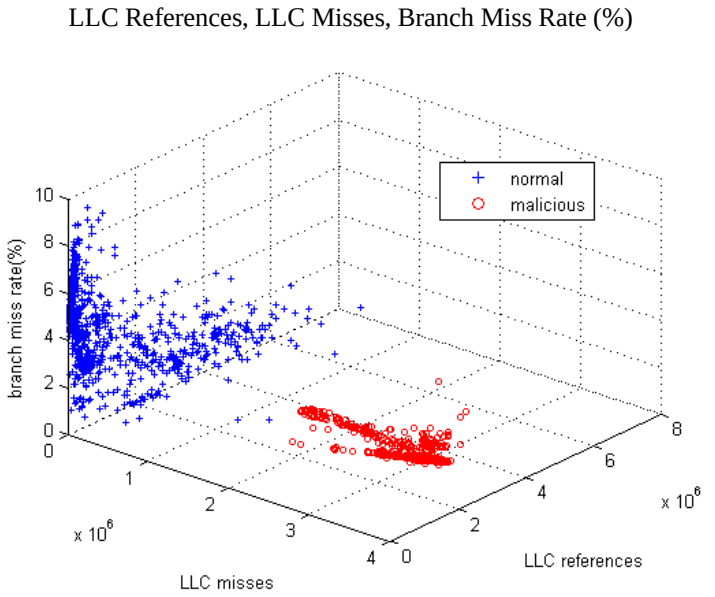

Figure 12. Distribution of LLC references, LLC misses and branch miss rate features for Rowhammer.

Similarly, Figure 13 shows the distribution of normal and malicious sample points for *Spectre* attacks using branch miss rate and LLC miss rate parameters. Figure 14 uses LLC references, LLC misses and branch miss rate. We can also see that there are clear boundaries between the two classes in both figures. Therefore, we believe we can detect *Spectre* attacks using the microarchitectural features we selected.

Branch Miss Rate, LLC Miss Rate

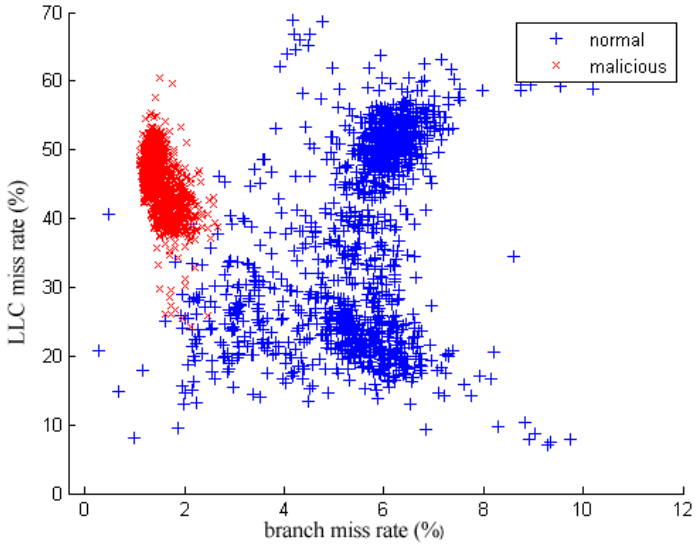

Figure 13. Distribution of branch miss rate and LLC miss rate features for Spectre.

LLC References, LLC Misses, Branch Miss Rate (%)

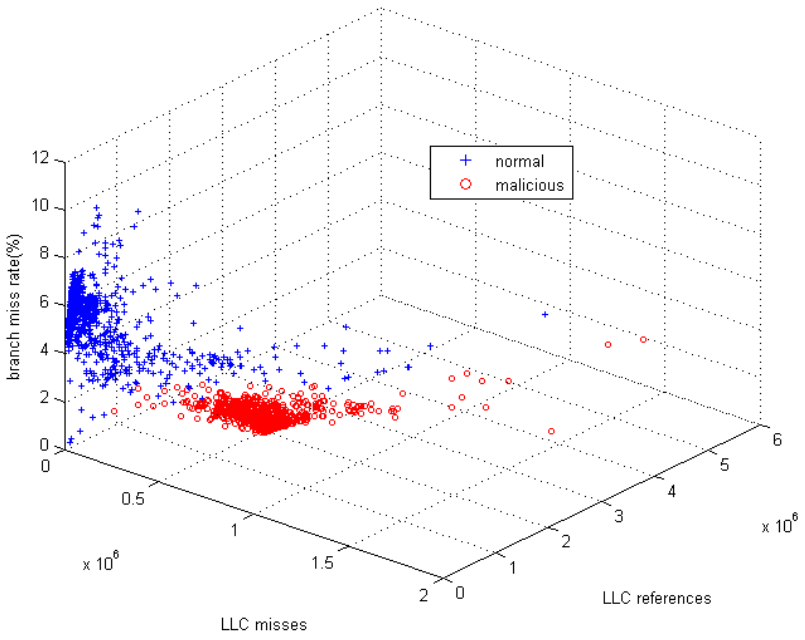

Figure 14. Distribution of LLC references, LLC misses and branch miss rate features for Spectre.

4.5.2 Online Detection Performance

Depending on the distribution of the collected performance counter data, we use different machine learning algorithms of different complexity. As mentioned in section 4.2.1, this is used

to train the base classifier. We then smooth the output time series with WMA and finally build the online detector based on the sliding window approach described earlier. The performance of the detection varies with the classifiers used.

A simple model using Logistic Regression is first built with default parameters. To further enhance the detection accuracy, different parameters are tuned for the model. If the theoretical best accuracy is not reached, we then move on to more complex models using Support Vector Machine, then Multi Layer Perceptron. We use randomized search over different parameters of different models to find the best combination, where each setting is sampled over a distribution of possible parameter values. Compared with exhaustive search, it is less computationally expensive and gives results that are close to the optimal solution. The parameters for different classifiers used in our experiments are as follows:

Logistic Regression (LR):

- *Regularization strength C*: Trades off misclassificaiton of training examples against simplicity of decision boundary. A smaller C gives a smoother boundary for stronger regularization,

- *Regularization parameters L1 and L2:* Prevents overfitting by imposing a penalty on the coefficients.

Support Vector Machine (SVM):

- *Gamma parameter*: Defines how far the influence of a single training sample reaches.

- *Regularization strength C*: Works similarly to the C paramenter in LR.

- *Kernel*: Includes linear, radial basis function (RBF), or polynomial kernel.

Multi Layer Perceptron (MLP):

- *Hidden layer sizes*: Defines the number of hidden layers in the network and number of hidden neurons in each layer.

- *Activation function*: Actives the neurons in the hidden layers, can be logistic, rectified linear unit function or hyperbolic tangent function.

- *Regularization parameter alpha*: Avoids overfitting.

To choose the most suitable classifier, in real world applications such those found in embedded systems, we need to consider constraints of time, power consumption, memory resources, etc. In addition, we need to know how much the system is allowed to tolerate in terms of false positives and false negatives.

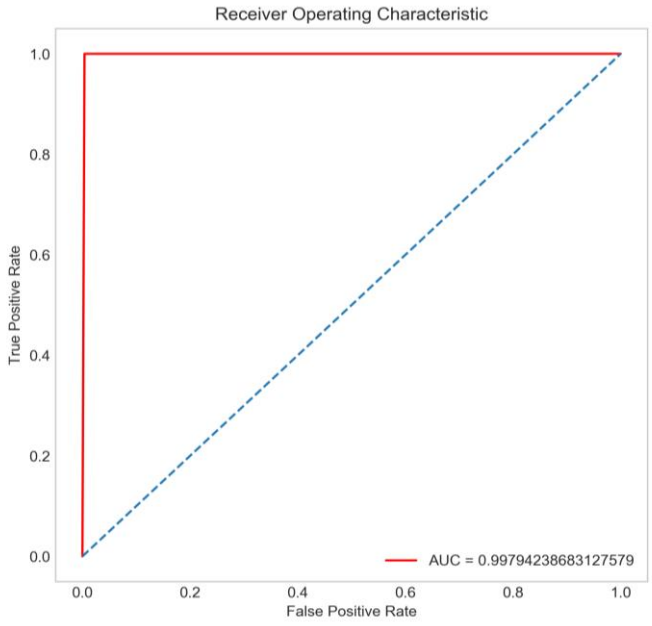

Figure 15. ROC for online detection of Rowhammer using logistic regression.

To quantitatively evaluate the performance of online detection based on different classifiers, we look at the Receiver Operating Characteristic (ROC) curves which plots false positive rate as the x-axis against true positives as the y-axis as shown in Figure 15, Figure 16 and Figure 17. Indeed, ROC curves are typically used to show the tradeoff between false positives and true positives. If we allow a higher rate of false positives (in other words, moving towards the right of the graph), the detector should be able to catch more malicious attacks. The dotted diagonal line connecting (0,0) and (1,1) represents the performance of a classifier that randomly guesses. For a classifier performs better than random guess, its ROC will lie above the diagonal. We can see that all our trained classifiers have better performance than random guess.

Figure 15 shows the ROC curve for online detection of *Rowhammer* attacks using simple Logistic Regression with default parameters. We can see the trade-off is minimal in this case: as we move from the top left corner of the ROC curve to the right along the curve to allow more false positives, the true positive rate does not increase much (from 99.77% to 100%). To choose the best configuration, we pick the point on the curve at the top left corner which gives the lowest sum of false negatives and false positives. For a simple model such as LR, we are able to obtain an AUC value of 0.9979 which is obviously very close to 1.

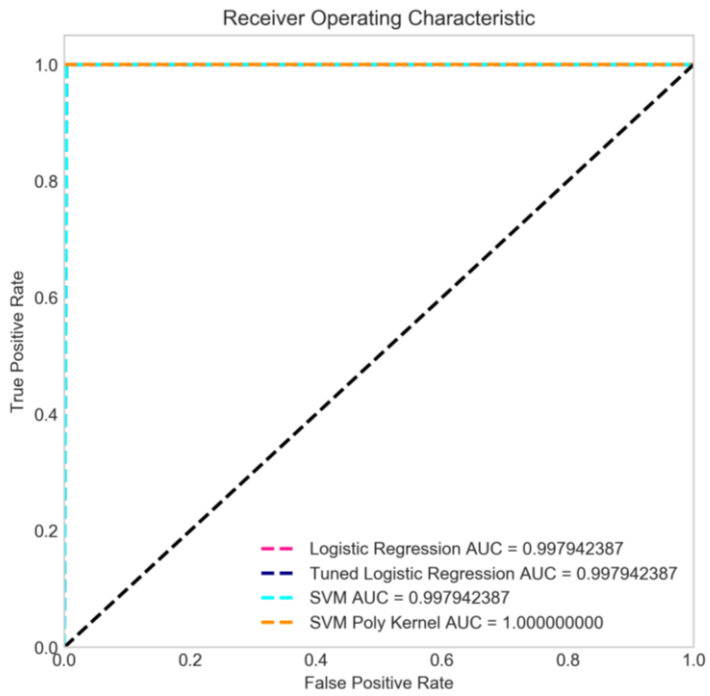

Figure 16. ROC for online detection of Rowhammer using different classifiers.

Figure 16 compares the ROC curves for all the trained classifiers for *Rowhammer* attacks in our experiment. We can see that they all perform very well with an AUC value above 0.99. As we were building models with increasing complexity, the SVM with polynomial kernel of degree 3 could reach the best possible AUC value of 1, which means a 0% error rate. Therefore, we conclude that increasing the complexity of models based on MLP would not lead to any further improvement.

Similarly, in Figure 17, we plot the AUC curves for the online detectors using different classifiers for *Spectre* attacks. Compared with Rowhammer attacks, the performance counter data are spread more randomly as we discovered while training and testing the classifiers with increasing complexities. Therefore, we built more complex models using MLP with 2 hidden layers. In general, all the classifiers give fairly good results with AUC values above 0.98. Overall, MLP outperforms other classifiers with the highest AUC value.

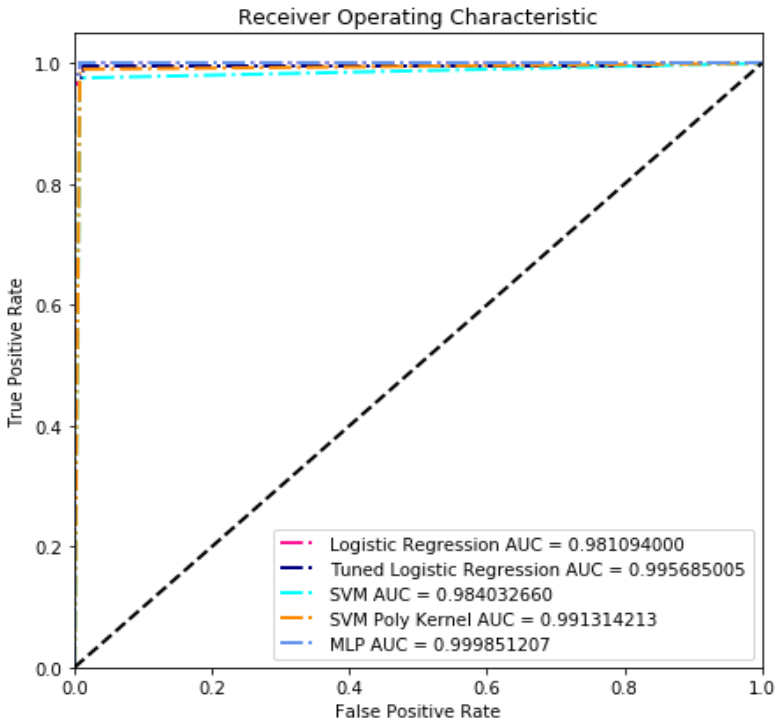

Figure 17. ROC for online detection of Spectre using different classifiers.

We compare the performance of each classifier quantitatively using the AUC index and we choose the best point on the ROC which gives the minimum FP and FN shown in Table 2 and Table 3. For *Rowhammer* attacks, using SVM with a polynomial kernel of degree 3 can already achieve perfect results (0% error rate). For *Spectre* attacks, the best case using MLP with 2 hidden layers gives 0% false negatives with only 0.77% false positives. We also observe that more complex models require longer training time since there are more parameters.

Table 2. Performance of Different Classifiers for Rowhammer.

Classifier	AUC	FP (%)	FN (%)	Training Time (sec)
LR	0.9979423868	0	0.23	0.03
Tuned LR	0.9979423868	0	0.23	0.03
SVM	0.9979423868	0	0.23	0.05
SVM with Polynomial Kernel	1	0	0	7

Table 3. Performance of Different Classifiers for Spectre.

Classifier	AUC	FP (%)	FN (%)	Training Time (sec)
LR	0.9810939999	3.83	3.40	0.04
Tuned LR	0.9956850054	1.15	2.43	0.04
SVM	0.9840326600	0.77	2.43	0.06
SVM with Polynomial Kernel	0.9913142134	0.77	0.97	9.8
MLP	0.9998512071	0.77	0	95

4.6 Evasion of Spectre Attack Detection

In this section, we study how the original *Spectre* mentioned in 4.1.4 could be even more maliciously updated to operate effectively without being detected by HPC-based classifiers. Previous research [91, 92, 93, 94] suggested generating evasive malware (also known as mimicry attack) by instruction insertion, code obfuscation, or calling of benign functions in between malignant payloads. Researchers [94] add instructions in the control flow graph of the malware in a way that does not affect the execution state of the program to evade HPC-based detectors. However, the inserted instructions may change the microarchitectural state such as the cache content of the victim. Compared with the above-mentioned evasive malware, developing evasive microarchitectural side channel attacks such as *Spectre* has additional requirements, because they are time sensitive and the attacker must ensure relevant microarchitectural status is unchanged in order to perform successful attacks. This research studies the feasibility of constructing evasive *Spectre* that is able to bypass HPC-based detector while maintaining a reasonable attack success rate, and also the trade-off between attack success rate and attack evasiveness. To achieve reasonable attack success rates, the attacker has to insert instructions or put the attack to sleep at

coarser granularity than a basic block in the control flow graph. Therefore, we define atomic tasks and reshape the microarchitectural features in the granularity of atomic task level. If an atomic task is interrupted, the attack success rate would be greatly reduced. We also quantitatively compare different strategies to determine the best way an attacker could use to evade detection while maintaining a reasonable success rate and speed.

4.6.1 Threat Model

We assume that the victim's machine is running an HPC-based malware detector such as proposed in section 4.2 to defend from Spectre attacks. The detector monitors four microarchitectural features including Last-Level Cache references (LLC references), Last-Level Cache misses (LLC misses), branch instructions retired (branches) and branch mispredict retired (branch mispredictions) at a fixed sampling rate on a separated core. In future research, the condition will be relaxed for mixed sampling rate and the detector can be implemented in dedicated hardware to reduce performance overhead. We assume the attacker's goal is to reveal some confidential memory content on the victim machine without being detected as malware. To achieve this goal, we further assume the attacker can observe the behavior of the malware classifier from a machine with a similar HPC-based detector as the victim machine. The attacker can evade detection by changing the microarchitectural characteristics of the updated Spectre so as to behave like benign programs.

We assume the attacker knows the features being monitored by the malware classifier. This is reasonable because the attacker knows that the original *Spectre* would cause increased cache misses and reduced branch mispredictions. However, the attacker does not know the sampling period of the detector. Yet this can be reverse engineered as demonstrated in [94].

As a Spectre attack runs in a loop, we assume the attacker can slow down the attack by calling the victim function at specific intervals. In addition, we assume the attacker can manipulate the performance counters by inserting instructions that reduce LLC cache misses and increase branch mispredictions by exploiting other vulnerabilities such as just-in-time code reuse attacks. This gives the attacker more privileges and could help us test the detector's resilience to evasion in extreme conditions.

Previous work [94] shows that the accuracy of HPC-based detectors decreases significantly as the number of instructions inserted in the original attack increases and assumes the attacker is interested in maintaining the performance of the attack. However, this did not consider how the inserted instructions may affect the success rate of the detector in inferring the correct content. Since *Spectre* is time sensitive as it utilizes a cache side channel to leak memory content, the inserted instructions may change or provide opportunities for other running programs to change the cache status and cause the attack to read the wrong content. Therefore, we further assume the attacker aims at maintaining a reasonable success rate.

4.6.2 Experimental Setup

We designed an attack on a machine similar to a typical victim laptop computer with Debian Linux 4.8.5 OS on an Intel Core i3-3217U 1.8 GHz processor with 3MB cache and 4GB of memory. We used the standard profiling infrastructure on Linux *perf* tools to obtain four performance counters data as discussed in section 4.6.1 including branch mispredictions, LLC misses, branches, and LLC references at each sampling interval.

In the clean environment, we sought to create realistic scenarios by browsing popular websites and streaming videos or running a text editor. For data collection when the system is under attack, we launched *Spectre* variant 1 proof of concept and "Evasive *Spectre*" attacks using the strategies proposed in section 4.6.4 on top of normal applications. The system status was reset after each run to ensure the measurements were independent across different runs.

4.6.3 Feasibility Analysis of Evasive Spectre

In order for the attack to be successful, the attacker has to complete malicious tasks faster than the detection frequency. Thus, the microarchitectural trace of the attack should be reshaped so the attack can make progress at each detection interval. We thus define an atomic task as a sequence of instructions that should not be interrupted during execution if progress is to be made towards the completion of a malicious task to achieve successful attack. We identified three *atomic tasks* in the proof of concept Spectre-V1: *(1) Flushing cache lines, (2) Mistraining branch predictor, (3) Attempting to infer the secret byte that is loaded into cache.* We compare the attack success rate of interrupting the attack during atomic tasks and between atomic tasks by inserting the same instructions. For each byte, the three tasks were performed multiple times to get the best results. The original attack read secret bytes at an approximate rate of 2KB/second on average.

As discussed in section 4.6.1, we assumed the attacker knows the features being monitored but does not know the classification period. we used the method proposed in [94], to collect multiple pairs of testing and training data sets of the same features using different collection periods and train a reverse engineered detector for each data set. The victim's

collection period (100ms) is the same as the collection period of the reverse-engineered detector with the highest accuracy.

Since the sampling period of the victim detector is much larger than the time taken to perform each atomic task, the attacker can transform the microarchitectural profile of *Spectre* by inserting instructions or "sleeping" at a finer granularity than the sampling rate of the detector. To analyze the feasibility of evading detection, we execute *atomic tasks* using 20% of each sampling period and put the attack to sleep for the remainder. Figure 18 shows the distribution of (1) benign, (2) malicious and (3) evasive sample points using cache miss rate and branch miss rate features. It shows a clear boundary between normal and malicious sample points while the evasive sample points shift the original malicious to overlap with normal ones (they cannot fully overlap because unlike normal programs, the evasive attack still needs to perform malicious tasks). Modified *Spectre* is performs with an 89% success rate. This shows the feasibility of constructing an "Evasive *Spectre*."

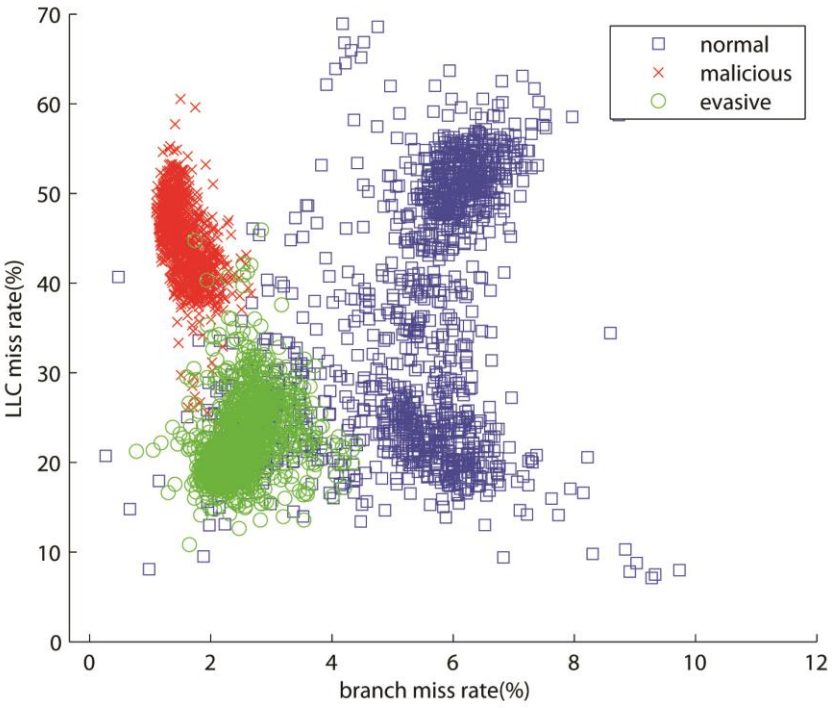

Figure 18. Branch miss rate vs. LLC miss rate for evasive Spectre.

4.6.4 Strategies to Construct Evasive Spectre

The original *Spectre* increases LLC misses and reduces branch mispredictions. To evade detection, the attack could be slowed by putting it to sleep or inserting instructions that reduce the number of LLC misses (reading the same memory bytes) and increase the number of branch mispredictions (adding unpredictable branches). Assuming the attack runs in a loop, at each cycle, the attacker needs to complete a series of atomic attacks to retrieve one secret byte from the victim. We thus considered the following four strategies:

1. Put the attack to sleep in between *atomic tasks*.

2. Put the attack to sleep after all tasks have completed.

3. Insert instructions in between *atomic tasks*.

4. Insert instructions after all tasks have completed.

The first two strategies slow down the attack and strategies 3 & 4 directly manipulate the performance counters.

Varying sleep time or looping the instructions that reshape the microarchitectural profile different times will accordingly change the attack bandwidth. We studied the effectiveness of different strategies by gradually reducing the attack bandwidth and analyzing the results in the following section.

4.6.5 Results

We now evaluate different evasion strategies proposed in section 4.6.4 by varying the bandwidth reduction from 1X to 7X and comparing the detection accuracy of the victim detector and the success rate of the attack. We define the success rate as the percentage of correct bytes inferred by the attacker over the total number of bytes inferred. Putting the attack to sleep or inserting instructions to reshape the microarchitectural profile of the attack reduces the rate of confidential content read. The longer the attack takes to reshape the profile, the more bandwidth reduction it will incur and the lower the success rate, due to higher TLB and cache pollution. Therefore, an effective evasion strategy should result in a low detection accuracy and maintain a reasonable attack success rate and bandwidth.

For each experimental setup with different evasion strategies and bandwidth reduction, we record the attack success rate and detection accuracy using the existing victim detector with different Machine Learning classifiers (Logistic Regression (LR), Support Vector Machine

(SVM), and Multi-Layer Perceptron (MLP)). We collect data in 10 independent runs for each setup and calculate the average values to avoid bias.

Figure 19 to Figure 22 show the detection accuracy over bandwidth reduction from 1X to 7X using different ML classifiers for each of the four evasion strategies. In all cases, the detection accuracy drops with the bandwidth because the attack becomes more evasive and closer to benign programs as it runs slower. In addition, the MLP classifier retains a better detection accuracy as the bandwidth drops. Therefore, MLP yields a higher resiliency to evasive attacks. In contrast, it is easier to avoid detection by a simple LR classifier.

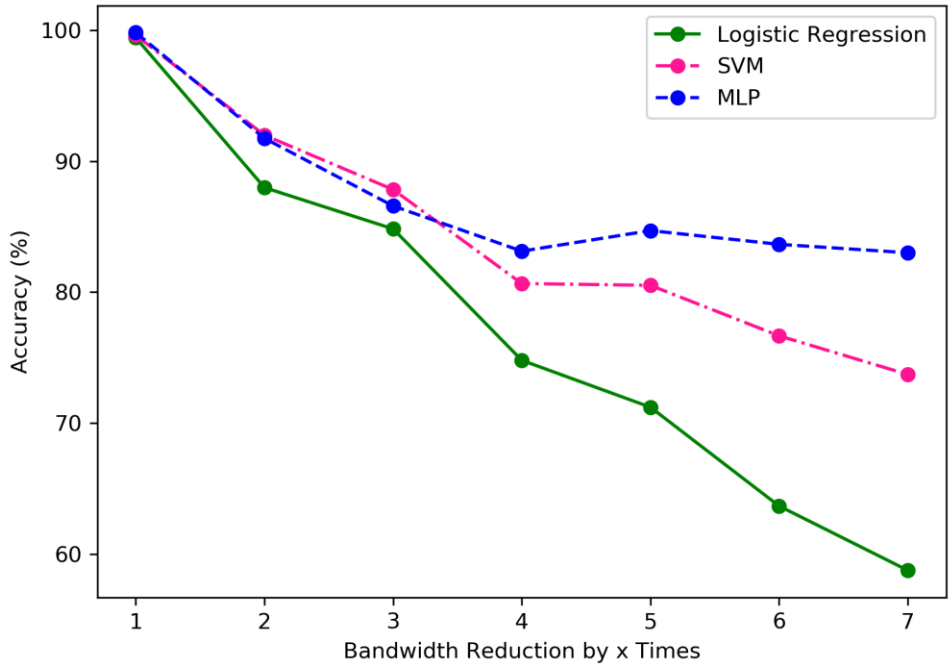

Figure 19. Detection accuracy for strategy 1 (sleep after all tasks).

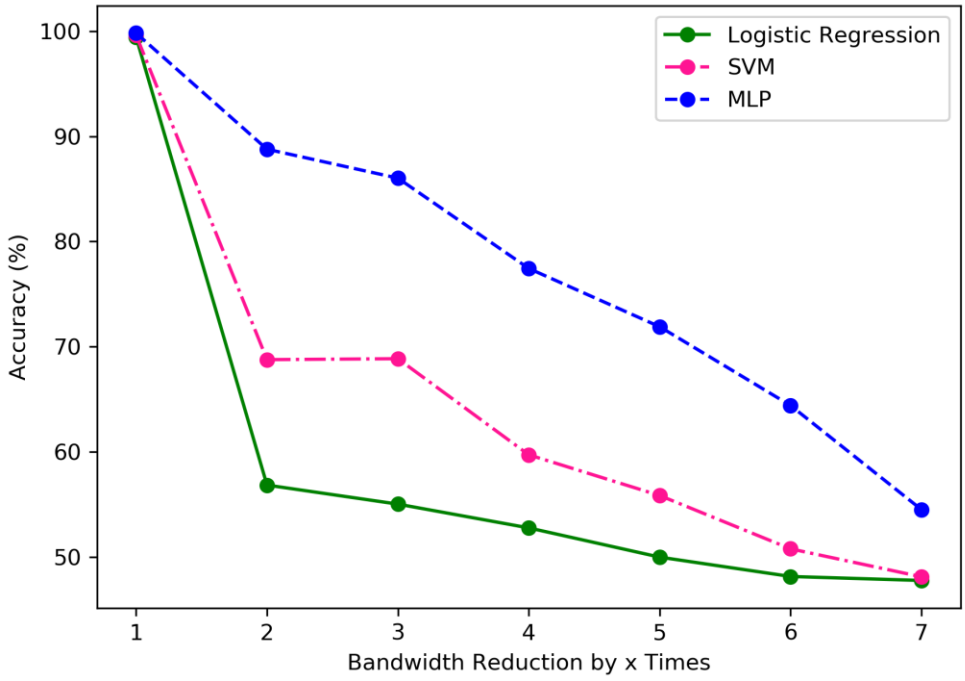

Figure 20. Detection accuracy for strategy 2 (sleep between atomic tasks).

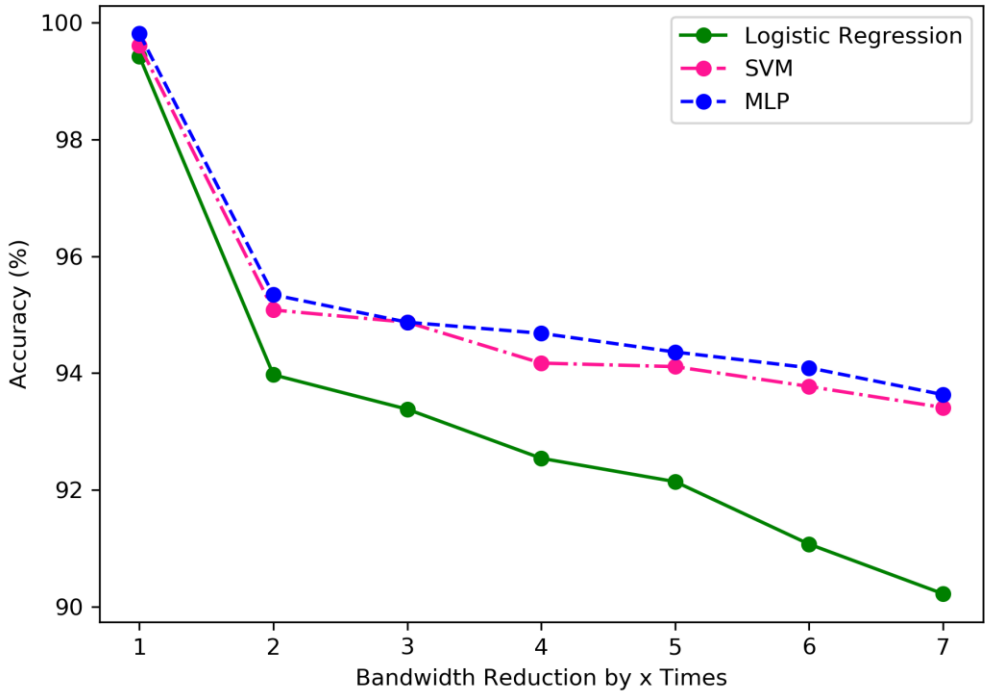

Figure 21. Detection accuracy for strategy 3 (insert instructions after all tasks).

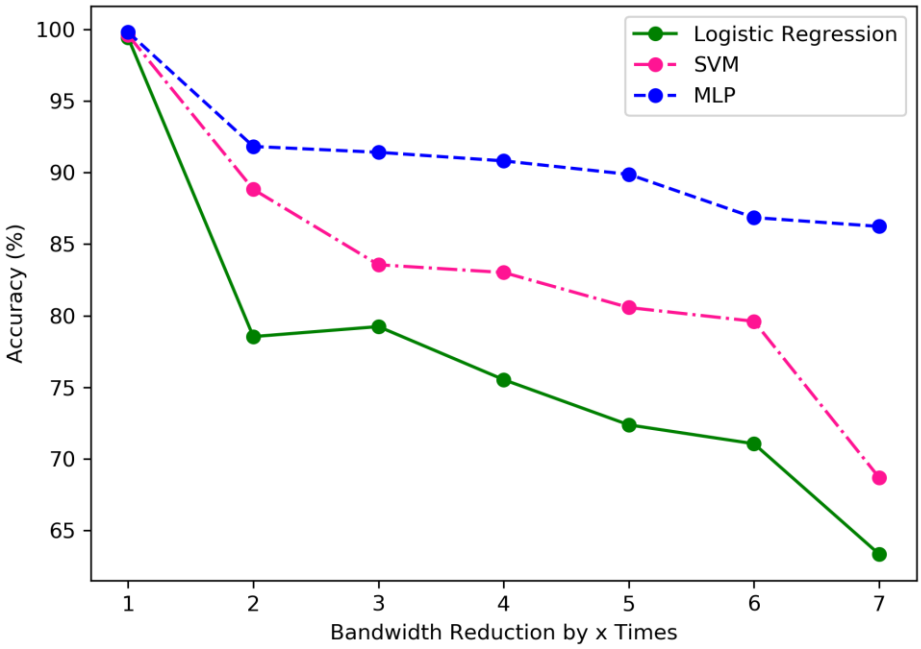

Figure 22. Detection accuracy for strategy 4 (insert instructions between atomic tasks).

Figure 23 to Figure 25 show for each classifier the detection accuracy of different evasion strategies as bandwidth decreases. For all, strategy 1 (put the attack to sleep in between atomic tasks) causes the detection accuracy to drop fastest as bandwidth drops. The detection accuracy diminishes to around 50% (random guess) when the bandwidth reduction is 7X. Therefore, strategy 1 produces the most evasive attack. On the other hand, strategy 4 (insert instructions after all tasks are done) performs the worst in this regard. Strategy 2 and 3 perform similarly in terms of evading detection. Note that shaping the microarchitectural profile in between atomic tasks yields a more evasive attack than shaping it after all the tasks are done no matter what method (sleep or insert instructions) is used.

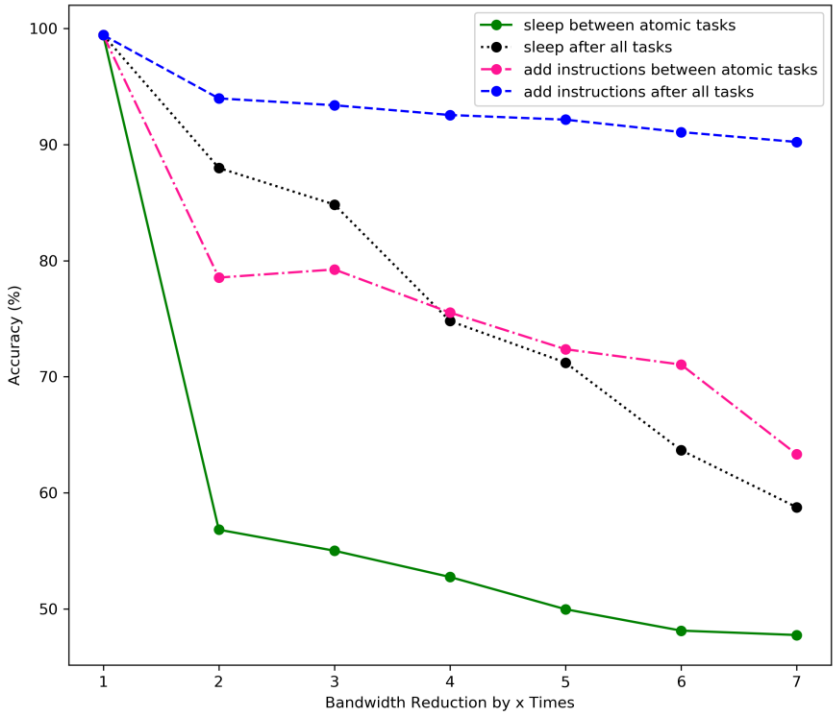

Figure 23. Detection accuracy using Logistic Regression.

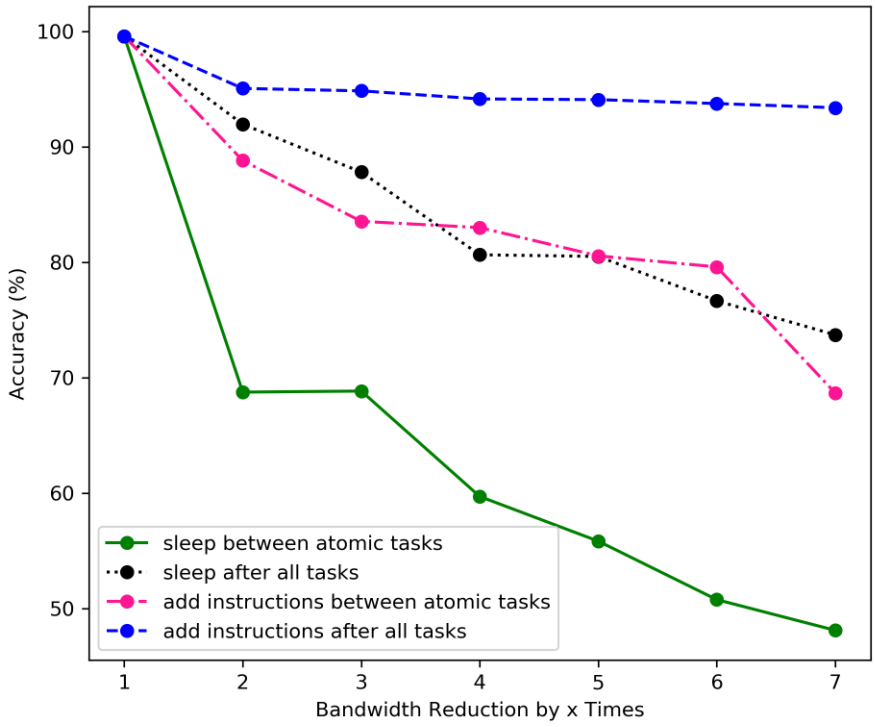

Figure 24. Detection accuracy using Support Vector Machine.

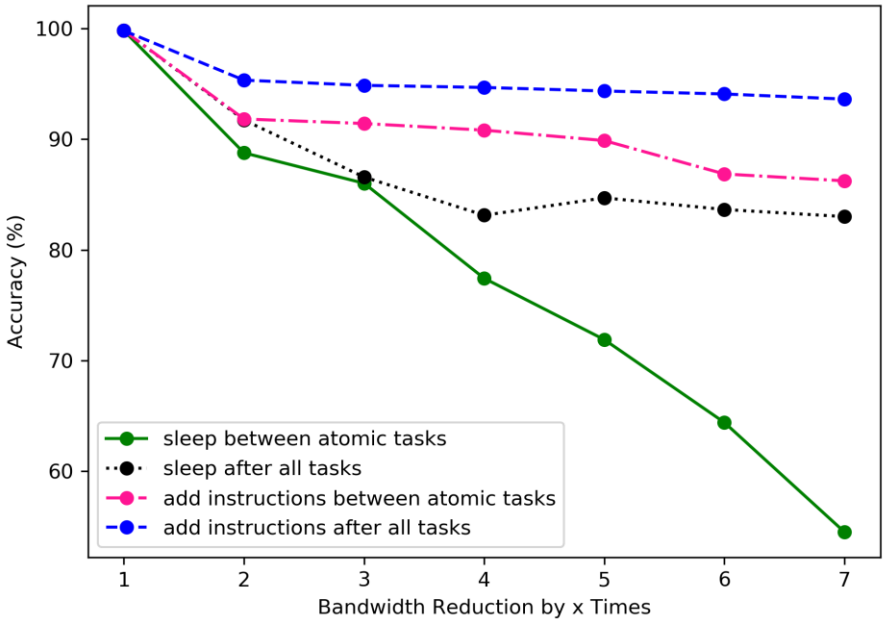

Figure 25. Detection accuracy using Multi-Layer Perceptron.

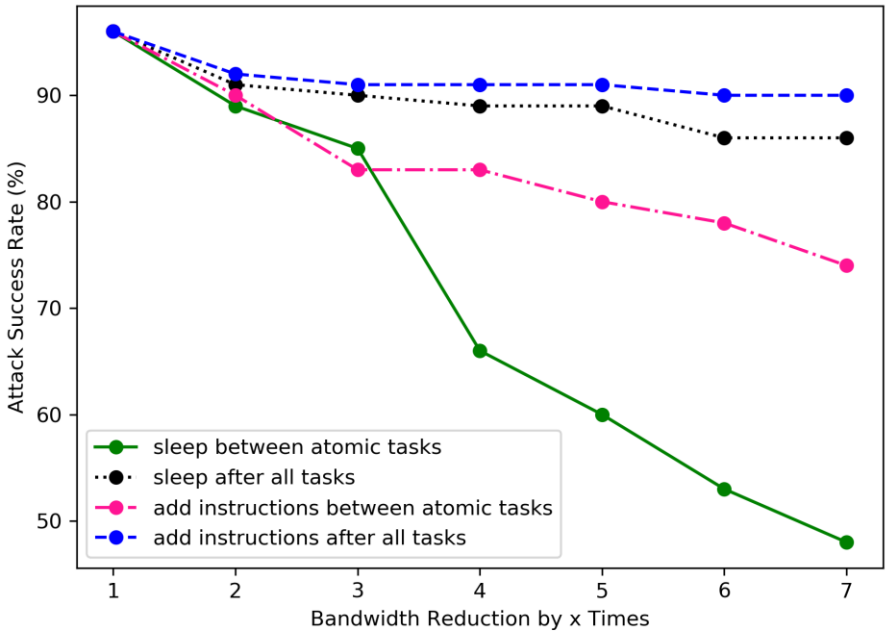

Figure 26. Attack success rate using the proposed evasion strategies.

Figure 26 shows the attack success rate using different evasion strategies. As the attack bandwidth decreases, so does the success rate. Therefore, the attack has a higher chance to fail

when it is running at a lower speed due to possible TLB and cache pollution by other processes. Similarly, inserting instructions or sleeping between atomic tasks has a higher chance to fail than inserting or sleeping after all tasks are done. The success rate drops below 50% with strategy 1 at 7X bandwidth reduction.

Inserting instructions or putting the attack to sleep more frequently may result in a lower success rate due to TLB and cache pollution. In order to avoid detection, the attack must run at lower bandwidth and success rate. Considering both detection accuracy and attack success rate, strategy 1 produces the most evasive attacks but the lowest success rate. Conversely, strategy 4 has the best success rate but is the least evasive. Strategy 2 gives slightly more evasive attacks than strategy 3 as bandwidth reduces further; strategy 2 has a better attack success rate than strategy 3. Therefore, strategy 2 is on the overall most effective. At a 7X bandwidth reduction, the LR classifier can only perform at 58.77% accuracy, i.e., no better than a random guess. At the same time, the attack success rate remains at 85%. Therefore, with strategy 2, the attacker can evade detection by an LR classifier without sacrificing much in terms of success rate and bandwidth. To evade the scrutiny of a more complex classifier such as MLP, the attacker can further reduce the bandwidth for a lower detection accuracy. At 10X bandwidth reduction, the MLP classifier performs at 70% accuracy and the attack success rate remains at 85%.

4.7 Conclusion

The impact of malicious attacks targeting hardware vulnerabilities can be catastrophic and widespread as they usually can bypass traditional software-based security defenses. We proposed to detect such attacks by monitoring microarchitectural features deviations. This is done by collecting related data from performance counters. We take *Rowhammer* (exploits

DRAM disturbance error vulnerability) and *Spectre* (exploits speculative execution and side channel vulnerabilities) attacks to demonstrate the feasibility and effectiveness to detect such attacks using microarchitectural features. The features are collected from hardware performance counters normally available in modern processors. An online detection method is adopted to detect malicious behaviors during the attack at early stage rather than offline detection after the damage is done. The experimental results show promising detection accuracy with 0% overall error rate for *Rowhammer* attacks using SVM, and only 0.77% false positives and 0% false negatives for *Spectre* attacks using a trained multilayer Perceptron classifier. As complete mitigation to the *Spectre* is challenging, it is imperative to dynamically detect such attacks.

There are many variants of *Spectre* depending on the types of hardware design flaws and side channels being exploited. New variants are continuously discovered, and researchers recently identified a new speculative store bypass vulnerability [34]. It should be noted however, that all the variants use a side channel to infer confidential information in the final stage of the attack. If attacks could be performed in some stealth mode so that there would be no effect on the microarchitectural trace, it would be hard to detect the attack using the proposed methods. However, we believe it is also extremely difficult to effectively design such attacks for several reasons: first because the attack itself is very time consuming: for each bit the attacker must perform multiple training rounds on the branch predictor followed by cache side channel attacks. Adding additional delays may make the attack extremely slow. Second, adding delays may make the cache side channel ineffective as other running programs may change the cache status during the delay. Third, as the attack is hardware specific, it is very difficult to design the attack in stealth mode that works for different machines. We thus conclude it is possible to detect malicious behaviors by monitoring changes in these hardware side channels. For *Rowhammer*

attacks, there are also many new variants that circumvent recent security defenses for different architectures ranging from mobile devices to cloud servers. The research we have just presented has shown that the proposed approach is particularly effective for *Rowhammer* and *Spectre* attacks targeting hardware vulnerabilities. Future work will examine other attacks which exploit other hardware design vulnerabilities in different domains such as CPU, memory, GPU, *etc*.

We have also demonstrated the feasibility of a re-written *Spectre* which evades HPC-based malware detectors and proposed strategies to reshape the microarchitectural profile of the attack by putting attacks to sleep or inserting instructions. We have shown that putting Spectre to sleep after it has performed malicious tasks allows an attacker to effectively evade simple LR malware classifiers and maintain as high a success rate as 86% with a concomitant 7X bandwidth reduction. Complex models such as MLP mean higher resiliency to evasion, however, with a 10X bandwidth reduction, and a lower 70% detection accuracy. More sophisticated detectors will be studied in the future. They can be used to counter evasion by using a higher or randomized sampling rate.

Chapter 5. Conclusion

5.1 Summary

Malware detectors protect computer systems by monitoring and detecting malicious behavior. In this dissertation, we emphasize on detecting abnormal behavior in programs through malware detection at the hardware level. In Chapter 2, the classification of conventional software-based malware detection techniques is summarized. Then we review some of the previous software-based detection techniques that were adapted and implemented in the hardware. Other novel detection techniques that utilized the hardware-level performance features are also discussed. In Chapter 3, we design and implemented a reconfigurable hardware accelerated malware detector using CFI. The detector in implemented on a tightly coupled FPGA to provide reconfigurability during runtime according to different running applications on the CPU. Compare to previous research, the system provides more flexibility with reduced performance overhead. In Chapter 4, we focus on detection of malicious attacks targeting hardware vulnerabilities. This class of attacks are increasing rapidly and cannot be detected by traditional antivirus. Our detector using performance counter data to monitor the anomalies in the architectural and microarchitectural behaviors of system and was demonstrated to detect novel attacks like Rowhammer and Spectre with high accuracy. We further shown that, similar to typical malware, microarchitectural side channel attacks such as Spectre can also be updated to evade existing protection. Future detector will need to be robust enough to counter such stealthy attacks.

5.2 Future Directions

To reduce the performance overhead of the performance counter based malware detection system we developed in Chapter 4, we will implement it in dedicated hardware similar to the one used in Chapter 3 and study its performance and area overheads. Instead of using an offline machine learning classifier, the detection system can be enhanced to be self-evolving using online learning algorithm taking feedbacks from the running system and end-user. On the other hands, different variants of Spectre and other novel attacks can be tested with our malware detectors. Malware targeting other system components such as GPU, and attacks targeting server systems can also be explored.

REFERENCES

[1] G. McGraw and G. Morrisett, "Attacking malicious code: A report to the infosec research council," IEEE Software, 17(5): 33–44, 2000.

[2] A. Vasudevan and R. Yerraballi, "Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation," In Proceedings of the 29th Australasian Computer Science Conference, pages 311–320, 2006.

[3] 2015 Internet Security Threat Report, Volume 20, available online http://www.symantec.com/security_response/publications/threatreport.jsp

[4] Vinod P., V.Laxmi, M.S.Gaur, "Survey on Malware Detection Methods," 3rd Hackers' Workshop 2009

[5] N. Weaver, V. Paxon, S. Staniford, and R. Cunningham, "A taxonomy of computer worms," In Proceedings of the 2003 ACM Workshop on Rapid Malcode, pages 11–18, 2003

[6] D. Wagner and D. Dean, "Intrusion detection via static analysis," IEEE Symposium on Security and Privacy 2001.

[7] A. Sung, J. Xu, P. Chavez, and S. Mukkamala. "Static analyzer of vicious executables (save)," In Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04), 00:326–334, 2004

[8] S. Kumar and Spafford E. H. "A generic virus scanner in c++," In Proceedings of the 8th Computer Security Applications Conference, pages 210 – 219, 1992.

[9] A. Sulaiman, K. Ramamoorthy, S. Mukkamala, and A.H. Sung. "Malware examiner using disassembled code (medic)." Systems, Man and Cybernetics (SMC) Information Assurance Workshop 2005, June 2005

[10] K. Ilgun, R. A. Kemmerer, and Porras P. A. "State transition analysis: A rule-based intrusion detection approach," IEEE Transactions on Software Engineering, 1995.

[11] D. Ellis, J. Aiken, K. Attwood, and S. Tenaglia. "A behavioral approach to worm detection," In Proceedings of the 2004 ACM Workshop on Rapid Malcode, pages 43–53, 2004.

[12] A. Mori, T. Izumida, T. Sawada, and T. Inoue. "A tool for analyzing and detecting malicious mobile code," In Proceedings of the 28th International Conference on Software Engineering, pages 831 – 834, 2006.

[13] K. Wang and S. J. Stolfo. "Anomalous payload-based network intrusion detection," In Proceedings of the 7th International Symposium on (RAID), pages 201–222, September 2004.

[14] W. Lee and S. Stolfo. "Data mining approaches for intrusion detection," In Proceedings of the 7th USENIX Security Symposium, 1998.

[15] S. Hofmeyr, S. Forrest, and A. Somayaji. "Intrusion detection using sequences of system calls," Journal of Computer Security, pages 151 – 180, 1998.

[16] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. "A fast automaton-based approach for detecting anomalous program behaviors," In IEEE Symposium on Security and Privacy, 2001.

[17] I. Sato, Y. Okazaki, and S. Goto. "An improved intrusion detection method based on process profiling," IPSJ Journal, 43:3316 – 3326, 2002.

[18] W. Li, K. Wang, S. Stolfo, and B. Herzog. “Fileprints: Identifying file types by n-gram analysis,” 6th IEEE Information Assurance Workshop, June 2005

[19] Y. M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. “Detecting stealth software with strider ghostbuster,” In Proceedings of the 2005 International Conference on Dependable Systems and Networks, pages 368–377, 2005.

[20] W. Masri and A. Podgurski. “Using dynamic information flow analysis to detect attacks against applications,” In Proceedings of the 2005 Workshop on Software Engineering for secure sytems – Building Trustworthy Applications, 30, May 2005.

[21] C. Ko, G. Fink, and K. Levitt. “Automated detection of vulnerabilities in privileged programs by execution monitoring,” In Proceedings of the 10th Annual Computer Security Applications Conference, pages 134–144, December 1994.

[22] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. “Protecting against unexpected system calls,” Usenix Security Symposium, 2005.

[23] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, and N. Tawbi. “Static detection of malicious code in executable programs,” Int. J. of Req. Eng., 2001.

[24] J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. “Static analysis of binary code to isolate malicious behavior,” In 8th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999.

[25] M. Debbabi, E. Giasson, B. Ktari, F. Michaud, and N. Tawbi. “Secure self-certified cots,” In Proceedings of the 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, pages 183–188, 2000.

[26] D. Wagner and D. Dean. “Intrusion detection via static analysis,” IEEE Symposium on Security and Privacy, 2001.

[27] M. Kuhn, “The trust No 1 cryptoprocessor concept,” Purdue Univ., (1997) [Online]. Available: http://www.cl.cam.ac.uk/ mgk25/

[28] S. W. Smith and S. H. Weingart, “Building a high-performance, programmable secure coprocessor,” in Proc. Computer Network, 1999, pp.831–860.

[29] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-Flow Integrity Principles, Implementations, and Applications,” Proc. ACM Conf. Computer and Comm. Security (CCS), pp. 340-353, Nov. 2005.

[30] D. Arora, S. Ravi, A. Raghunathan, and N.K. Jha, “Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 14, No. 12, December 2006.

[31] S. Mao and T. Wolf, “Hardware Support for Secure Processing in Embedded Systems”, IEEE Transactions on Computers, Vol. 59, No. 6, June 2010.

[32] R.G. Ragel, S. Parameswaran, and S.M. Kia, “Micro Embedded Monitoring for Security in Application Specific Instruction-Set Processors,” Proc. 2005 Int’l Conf. Compilers, Architectures, and Synthesis for Embedded Systems (CASES), pp. 304-314, Sept. 2005.

[33] T. Zhang, X. Zhuang , S. Pande and W. Lee, “Anomalous path detection with hardware support,” in Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, San Francisco, California, USA, September 24-27.

[34] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. “MoCFI: A framework to mitigate control-flow attacks on smartphones,” In Network and Distributed System Security Symposium, NDSS ’12, 2012.

[35] Jannik Pewny, Thorsten Holz, “Control-flow restrictor: Compiler-based CFI for iOS,” In Annual Computer Security Applications Conference, ACSAC ’13, 2013.

[36] V. Pappas, M. Polychronakis, and A. D. Keromytis. “Transparent ROP exploit mitigation using indirect branch tracing,” In USENIX conference on Security, SSYM’13, 2013.

[37] Y. Xia, Y. Liu, H. Chen, and B. Zang. “CFIMon: Detecting violation of control flow integrity using performance counters,” In Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN ’12, 2012.

[38] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi, “Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation”, 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, June 2014.

[39] http://blogs.msdn.com/b/vcblog/archive/2014/12/08/visual-studio-2015-preview-work-in-progress-security-feature.aspx, 2014.

[40] S. Forrest, A. Somayaji, and D. Ackley. “Building diverse computer systems,” In Hot Topics in Operating Systems, 1997.

[41] S. Bhatkar, R. Sekar, and D. C. DuVarney. “Efficient techniques for comprehensive protection from memory error exploits,” In USENIX Security Symposium, 2005.

[42] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. “Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software,” In Annual Computer Security Applications Conference, 2006.

[43] V. Pappas, M. Polychronakis, and A. D. Keromytis. “Smashing the gadgets: Hindering return-oriented programming using in-place code randomization,” In IEEE Symposium on Security and Privacy, 2012.

[44] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization,” In USENIX Security Symposium, 2012.

[45] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. “ILR: Where’d my gadgets go?,” In IEEE Symposium on Security and Privacy, 2012.

[46] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, “Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code,” In ACM Conf. on Computer and Communications Security, 2012.

[47] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, “Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization”, IEEE Symposium on Security and Privacy, 2013

[48] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, Yier Jin “HAFIX: Hardware-Assisted Flow Integrity Extension”, ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, 2015.

[49] L. Davi, D. Lehmann, A. R. Sadeghi, and F. Monrose. “Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection,” In USENIX conference on Security, SSYM’14, 2014.

[50] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. “Out of control: Overcoming control-flow integrity,” In IEEE Symposium on Security and Privacy, S&P ’14, 2014.

[51] 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,” in Proceedings of the 40th Annual International Symposium on Computer Architecture, ser. ISCA ’13. New York, NY, USA: ACM, 2013, pp. 559–570.

[52] A. Tang, S. Sethumadhavan, S. Stolfo, “Unsupervised anomaly-based malware detection using hardware features,” in Research in Attacks, Intrusions and Defenses, ser. Lecture Notes in Computer Science, 2014, vol. 8688, pp. 109–129.

[53] Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., Ponomarev, D.: “Malware aware processors: a framework for efficient online malware detection,” In: Proceedings of the International Symposium on High Performance Computer Architecture (HPCA) 2015.

[54] Khaled N. Khasawneh, Meltem Ozsoy, Caleb Donovick, Nael Abu-Ghazaleh, Dmitry Ponomarev, “Ensemble Learning for Low-Level Hardware-Supported Malware Detection,” RAID 2015, LNCS 9404, pp. 3–25, 2015.

[55] Cisco, “The Internet of Things”, http://share.cisco.com/internet-of-things.html

[56] G. McGraw and G. Morrisett, “Attacking malicious code: A report to the infosec research council. IEEE Software”, 17(5):33–44, 2000.

[57] A. Vasudevan and R. Yerraballi, “Spike: Engineering malware analysis tools us- ing unobtrusive binary-instrumentation”, In Proceedings of the 29th Australasian Computer Science Conference, pages 311–320, 2006.

[58] PCWorld, “Report: Average of 82,000 new malware threats per day in 2013”, http://www.pcworld.com/article/2109210/report-average-of-82-000-new-malware-threats-per-day-in-2013.html

[59] CVE Details, “Vulnerabilities By Type” (http://www.cvedetails.com/vulnerabilities-by-types.php)

[60] Vinod P., V.Laxmi,M.S.Gaur “Survey on Malware Detection Methods”, 3rd Hackers' Workshop 2009

[61] D. Wagner and D. Dean, “Intrusion detection via static analysis. IEEE Symposium on Security and Privacy”, 2001.

[62] M. Kuhn, “The trust No 1 cryptoprocessor concept,” Purdue Univ., (1997) [Online]. Available: http://www.cl.cam.ac.uk/ mgk25/

[63] S. W. Smith and S. H. Weingart, “Building a high-performance, programmable secure coprocessor,” in Proc. Comput. Netw., 1999, pp.831–860.

[64] D. Arora, S. Ravi, A. Raghunathan, and N.K. Jha, “Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 14, No. 12, December 2006

[65] S. Mao and T. Wolf, “Hardware Support for Secure Processing in Embedded Systems”, IEEE Transactions on Computers, Vol. 59, No. 6, June 2010

[66] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-Flow Integrity Principles, Implementations, and Applications,” Proc. ACM Conf. Computer and Comm. Security (CCS), pp. 340-353, Nov. 2005.

[67] R.G. Ragel, S. Parameswaran, and S.M. Kia, “Micro Embedded Monitoring for Security in Application Specific Instruction-Set Processors,” Proc. 2005 Int’l Conf. Compilers, Architectures, and Synthesis for Embedded Systems (CASES), pp. 304-314, Sept. 2005.

[68] T. Zhang, X. Zhuang, S. Pande and W. Lee, "Anomalous path detection with hardware support," in Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, San Francisco,california, USA, September 24-27.

[69] 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,” Proc. IEEE Fourth Ann. Workshop Workload Characteriza- tion, Dec. 2001.

[70] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard,P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In Proceedings of the 27th USENIX Security Symposium, pages 973–990,2018.

[71] Horn, J., “Reading privileged memory with a side-channel,” https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[72] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y., “Spectre attacks: Exploiting speculative execution,” ArXiv e-prints, Jan. 2018.

[73] D. J. Bernstein, “Cache-timing Attacks on AES”, http://cr.yp.to/antiforg ery/cachetiming20050414.pdf, 2005.

[74] Kim, Y.R., Daly, J., Kim, C., Fallin, J., Lee, H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors,” ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, USA, 14–18 June 2014.

[75] Seaborn M. and Dullien T., “Exploiting the DRAM rowhammer bug to gain kernel privileges,” in Black Hat Briefings, 2015.

[76] Razavi, K., Gras, B., Bosman, E., Preneel, B., Giuffrida, C., Bos, H., “Flip Feng Shui: Hammering a Needle in the Software Stack,” in the Proceedings of the 25th USENIX Security Symposium, 2016.

[77] Xiao, Y., Zhang, X., Zhang, Y., Teodorescu, M.R., “OneBitFlips, OneCloudFlops: Cross-VM Row Hammer Attacks and Privilege Escalation,” in the Proceedings of the 25th USENIX Security Symposium (SEC), 2016.

[78] Bosman, E., Razavi, K., Bos, H., Giuffrida, C., “Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector,” IEEE Symposium on Security and Privacy (SP), 2016.

[79] Gruss, D., Maurice, C., Mangard, S., “Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript,” International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2016.

[80] Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T., “ANVIL: Software-Based Protection Against Next Generation Rowhammer Attacks,” in the Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016.

[81] Qiao, R., Seaborn, M., “A new approach for rowhammer attacks,” 2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp.161–166 , May 2016.

[82] Duda R, Hart P, Stork D., “Pattern classification. 2nd ed,” New York: Wiley/Interscience; 2000.

[83] J. Frank, “Machine learning and intrusion detection: Current and future directions,” in Proc. National 17th Computer Security Conference, Washington,D.C., October 1994.

[84] Cristianini N, Shawe-Taylor J., “An introduction to support vector machines and other kernel-based learning methods,” Cambridge: Cambridge University Press; 2000.

[85] Schölkopf B, Smola A., “Learning with kernels: support vector machines, regularization, optimization, and beyond,” Cambridge, MA: MIT Press; 2002.

[86] Sharma, D.K., Sharma, H.P. & Hota, H.S., “Future Value Prediction of US Stock Market Using ARIMA and RBFN,” International Research Journal of Finance and Economics (IRJFE), 2015, 134, 136-145.

[87] Handa, R., Hota, H.S., & Tandan, S.R., “Stock Market Prediction with various technical indicators using Neural Network techniques,” International Journal for research in Applied Science and Engineering Technology (IJRASET), 2015, 3(4), 604-608.

[88] E. Keogh, S. Chu, D. Hart, M. Pazzani, “Segmenting time series: A survey and novel approach,” Data Mining Time Series Databases, vol. 57, pp. 1-22, 2004.

[89] Vafaeipour, M., Rahbari, O., Rosen, M.A., Fazelpour, F. & Ansarirad, P., “Application of sliding window technique for prediction of wind velocity time series,” International journal of Energy and environmental engineering (springer), 5,105-111, 2014.

[90] D. A. Osvik, A. Shamir and E. Tromer, “Cache attacks and Countermeasures: The Case of AES”, Cryptology ePrint Archive, Report 2005/271, 2005.

[91] K. Wang, J. J. Parekh, and S. J. Stolfo, “Anagram: A content anomaly detector resistant to mimicry attack,” in Proceedings of the9th International Conference on Recent Advances in Intrusion Detection. Springer-Verlag, 2006, pp. 226–248.

[92] D. Bruschi, L. Cavallaro, and A. Lanzi, “An efficient technique for preventing mimicry and impossible paths execution attacks,”in2007 IEEE International Performance, Computing, and Communications Conference, April 2007, pp. 418–425.

[93] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, “Scrap: Architecture for signature-based protection from code reuse attacks,” in 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), Feb 2013, pp. 258–269.

[94] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, “RHMD: Evasion-Resilient Hardware Malware Detectors,” in Proceedings of the Annual International Symposium on Microarchitecture, MICRO, Oct 2017, pp. 315–327.