공학박사학위논문

Extraction of Host Internal Information for External Hardware Security Monitors

하드웨어 기반 보안 모니터링을 위한 호스트 시스템의 정보 추출

2016년 2월

서울대학교 대학원
전기컴퓨터공학부
이 진 용

Abstract

Defending electrical devices against a variety of attacks is a daunting task. A lot of researchers have endeavored to address this issue by proposing security solutions that can attain high level of security while minimizing performance overhead introduced to the system. Among them, hardware-based security solutions have been noted for high performance compared to their software-based counterparts. However, we have witnessed that these mechanisms have rarely been accepted to the market. This phenomenon may be attributed to the fact that most solutions incur non-negligible modifications to the host architecture internals and thus would substantially increase the design time and manufacturing cost. In order to answer this problem, a hardware-based external monitoring has recently been proposed. The crux of this solution is that, being located outside the host core and connected to the host via a standard bus interface, the external monitor can efficiently conduct time-consuming monitoring tasks on behalf of the host while requiring no alteration to the host internals. However, these approaches either suffer from the incapability of handling various security problems or experience unsubtle performance overhead because, being externally placed and having no dedicated communication channels, the hardware monitor has a limited access to the information produced by the host core, and consequently, the system may be forced to use memory regions or other shared hardware resources to explicitly transfer the information from the host to the monitor hardware. In this thesis, we propose a security solution that can carry out more complicated security tasks with low performance overhead while keeping the host internal architecture intact. This can be archived by using an existing standard debug interface, readily available in numerous modern processors, to connect our security monitor to the host processor. In order to show the validity of our approach and explore the implication of using the debug interface for security monitoring, we present three security monitoring systems each of which addresses one of three well-known security issues: defending against kernel rootkits, tracking information-flow, and defense of code-reuse attacks. The experiment results show that, when implemented on a FPGA prototyping board, our monitoring solutions successfully detect the attack samples (i.e., data leakage attacks and CRAs). More importantly, our systems can attain significantly low performance overhead compared to previously proposed security monitoring solutions. The experiments also reveal that the area overhead of the hardware is acceptably small when compared to the normal sizes of today's mobile processors.

Keywords : Security, Hardware-based Security Monitoring, Core Debug Interface

Student number : 2011-30250

Table of Contents

Figures

Tables

Chapter 1

Introduction

As technology rapidly evolves, we are nowadays surrounded by a myriad electrical devices, including desktops, laptops, tablets, mobile phones, and smart watches. With these devices, we can do various things; they include seemingly trivial tasks such as sending emails and accessing social network sites to considerably more serious business like online banking and measuring medical information. As the amount of private information managed in these devices drastically increases, they are becoming more appealing targets of attackers.

In order to protect these devices (which we call the *system* from now on) against various attacks, a number of *security solutions* have been proposed in literature [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. Techniques they employ in their solutions can roughly be categorized as follows: (1) *encryption* (2) *obfuscation* (3) *isolation* (4) *system monitoring*. Encryption is a process of converting one data form into another, called *ciphertext*, basically making the original data impossible to be read or be understood by unauthorized users. Unlike encryption techniques that use *keys* to convert the whole data, obfuscation deliberately changes the layout of the structure of original data or inserts redundant logics to original source code in a way that malicious users cannot easily understand the original meaning of the code/data. Isolation is, as the name implies, the way of protecting information by placing it in a reserved, isolated space that cannot be accessed without proper authority. Lastly, system monitoring is an act of watching a specific set of system behaviors and checking whether the behaviors deviate from the ones of the legitimate system.

More specifically, the system monitoring technique mainly checks the execution behaviors of a program running on the host system to find any symptom of attacks. For monitoring, users or system administrators firstly define a set of *rules* (or *invariants*) that mostly signify the legitimate behaviors of target applications during normal code execution. At runtime, a security *monitor* deployed on the system checks whether there is any violation of these rules. If there is any detection of such an event, then the system administrators are notified of a possible attack.

Among many techniques, this system monitoring techniques have been adopted in many security solutions addressing a variety of attacks from monitoring the integrity of OS kernels [10, 12, 13, 14, 8, 9, 11] to monitoring application behaviors to detect any suspicious activities, for instance, by performing *dynamic information flow tracking* or *control flow integrity checking* [15, 16, 17, 18, 19, 20, 21]. These security monitoring schemes can be implemented in various forms. Two mainstream research directions are *software-based* [14, 8, 9, 22, 23, 15, 16, 17] and *hardware-based* approaches [7, 13, 10, 12, 11, 18, 19].

In general, the former approaches have popularity in the security community as they do not necessitate underlying hardware modification, thus allowing the easy deployment of such solutions to commodity hardware platforms. Moreover, they can easily adapt to new types of attacks as their core algorithms exist in the form of software, which can be readily updated in the field. Not surprisingly, however, they have a drawback in that they may impose substantial computational loads upon the host machine, which is not desirable for these solutions to be deployed in practice. For instance, in the case of DIFT, the flow of critical data needs to be monitored at every instruction that possibly moves the data from a secure storage to some other places. Even after aggressive optimizations [16, 17], the overhead still remains one or two orders of magnitude higher than that of hardware approaches [18, 19] in which extra hardware for monitoring operations is designed and integrated into an existing processor for acceleration. The main source of such a performance loss is that these software-based monitors are another software layer that competes with other software in the system for computing resources.

The hardware-based approaches, on the contrary, utilize an isolated hardware module which is physically independent of the monitored host system, thus unaffecting the operation of the tasks running on the host. Several works introduced in [13, 24, 25, 10, 12, 11, 26, 27, 28, 29, 30, 18, 19]. These hardware solutions tend to exhibit high performance by accelerating the monitoring process with the assistance of customized hardware logics for the task.

Despite their dramatic performance enhancement, they have a few drawbacks. The first one is that they may require the redesign of the existing processor architecture if one wants to maximize the overall performance while attaining high level of security. For instance, authors in [27, 28, 29] present solutions in which their hardware logics are tightly coupled with the host CPU for close monitoring of every control transfer during code execution. Such a close coupling requires major modifications to processor internal components such as pipeline datapaths or the structure of registers[31, 27, 28, 29], which would stymie the direct deployment of these solutions into commercial platforms.

The second drawback is that, if one chooses to avoid the aforementioned problem and places the monitor at the outside the host procesor as proposed in [12, 10, 11, 13, 20, 21], they may either have their system experiencing non-negligible performance overhead or have their monitor less useful than it should be. Undoubtedly, in their approaches, the host processor can concentrate on the execution of its own code while the time-consuming monitoring task is offloaded to the specialized hardware module outside the processor; in the literature, they empirically demonstrated that their monitoring scheme can be carried out in a great speed by external hardware, relieving significant burden for the extra computation from the host.

Nevertheless, there still remains an inefficiency. It originates from the limited ability of an external module to watch every internal state change dynamically made by the code running on the host. For precise security monitoring, the external monitor should be able to receive from the host various runtime information such as branch targets, memory addresses and register moves. Without such information, the effectiveness of such hardware monitors are subtatially reduced, and are allowed to perform only simple tasks such as monitoring memeoy access patterns as proposed in [13, 10, 11]. To conduct more sophisticated monitoring tasks, researchers in [21, 20] sugest the use of share memory regions to explicitly deliver the essential information from the host to the monitor. Even though their approaches exhibit a substantial performance improvement compared to software-based approches, there still remains non-negligible performance overhead mainly due to the tremendous amount of traffic for communication. As reported in [21, 20], the overehad is up to 30% of the total execution time even after all their optimizations through hardware communication buffers and special instructions.

In this thesis, we propose a new security *monitoring solution* that can resolve the problems of the previous works; the monitoring system implemented under the guidance of our solution can carry out more complicated security tasks with negligible performance overhead while keeping the host internal architecture intact. Complex as it sounds, but our monitoring solution merely suggests that the monitoring engine be placed externally to the host processor, similar to those introduced in [21, 20, 12, 10, 13], and that the engines be connected to the host processor via an existing standard interface, called the *core debug interface* (CDI), which is readily available for debugging purposes in various modern processors including ARM Cortex series and Intel x86 architectures [32, 33, 34].

Once they are plugged into CDI, our monitor engines can access bountiful information transmitted in the form of signals from CDI. The signals from CDI can largely be categorized into two groups. The first group of signals convey information relavant to the control-flow transfer of code executed on the host system. The second group of signals present memory access behaviors such as target addresses of a memory access with its corresponding (written/read) value for the access. In our solution, these groups of signals, which we hereafter call the *control-flow information* and *data-flow information* respectively, are the key element that enable us to build a variety of security monitoring mechanisms on external hardware monitors.

Simple as it may sound, but using these signals for building up a security system can involve several complications majorly because the initial set of signals from CDI cannot simply be fed into the security monitor as they are in their present form. Some signals originally generated for debugging may need to be translated into another form that is required for security monitoring. Therefore, in order to show the implication of using CDI for security monitoring, this thesis presents in its main chapters three well-known security problems, each of which monitoring solution requires different groups of signals to address the corresponding problem. In Chapter 3, attacks that target the integrity of an OS kernel is presented. The chapter shows, under the vigilance of our proposed monitor and thanks to the data-flow information transferred via CDI, no attack launched on the system can compromise the kernel. Chapter 5 describes a well-known attack technique that attackers tweak the control flow of a legitimate code into performing malicious operations. The second monitorng system presented in this chapter uses the control-flow information coming from CDI to detect any suspicious control-flow change. Chapter 4 presents a monitoring system that uses both control-flow and data-flow information to monitor the trace of secret data. At runtime, every data derived from the secret data is tainted and tracked throughout the operation, and if any tainted data is involed in potentially illegal activities, our monitoring system raises an alarm.

This thesis is organized as follows: Chapter 1 introduces and sumarizes the thesis. Chapter 2 provides a brief explanation of CDI and presents past work in the area of software-based and hardware-based monitoring solutions. Chapter 3 describes a monitoring solution that makes use of data-flow information to detect attacks targetting OS kernels. In Chapter 5, a monitoring solution that uses the control-flow information is explained. Chapter 4 shows the third monitoring solution that, to detect the misuse of critical data, makes use of both control-flow and data-flow information. And finally, Chapter 6

Chapter 2

Background and Related Work

2.1 Background

2.1.1 Core Debug Interface

As the complexity of software running on a processor drastically increases, recent commodity processor vendors tend to include rich debug features in thier hardware. The *On-chip debug* (OCD) unit [35, 34] is such a hardware feature that supports efficient real-time debugging/tracing without affecting the performance of the target processor. Provided by OCD, a rich set of information allows developers (or users), on their debugging environment (usually on desktop machines), to follow the path that the target CPU takes as a result of code execution and monitor values in various registers and memories. Representative examples of OCD are the ARM CoresSight modules [32] supported in ARM Cortex series processors such as the *embedded trace macrocell* (ETM) and the *program trace macrocell* (PTM).

CDI is an interface placed on the CPU side, whose main role is to provide OCD with the CPU's internal status information that is essential for debug/trace. In general, the OCD modules provide various signals to

Signal	Description
ETMICTL [20:0]	ETM instruction control bus
ETMIA [31:1]	ETM instruction address
ETMDCTL [10:0]	ETM data control bus
ETMDA [31:0]	ETM data address
ETMDD [63:0]	ETM data write data value
ETMCID [31:0]	Current processor Context ID

table 1: Description of CDI signals for ETM

developers such as instruction address, current context ID (or process ID), and data address/value of memory access instructions, which are useful information to keep track of the behavior of a monitored program. Thus, CDI for the OCD modules also provide such information through the dedicated signal lines [32]. As an example, in Table 1, the signals to ETM provided by ARM processors through CDI is described. Although the types of information supported by CDI can vary from one processor architecture to another, the signals presented in Table 1 are generally provided in most CDIs.

2.2 Related Work

As introduced in Chapter 1, there have been much research conducted to defend computer systems from various security attacks. In this section, we present a number of security solutions which mainly rely their detection techniques on monitoring the behaviors or patterns exhibited on the host system as a result of target program code execution. Starting from the description of the software-based monitoring solutions, we revisit past work on hardware-based monitoring solutions, and finally we address some work relavant to the use of debug interfaces to other purposes than their original usages.

2.2.1 Software-based Monitoring solutions

To defend the computer systems from various security attacks, there have been proposed a number of monitoring techniques, such as DIFT [22], memory bound checking and control flow integrity checking [15]. The most popular way to realize security monitoring schemes is to implement them in software. Most software monitoring approaches [22, 16, 36, 17, 37, 15] have relied upon either *source-code instrumentation* or *dynamic binary translation* (DBT) [38] for the defense against diverse attacks at execution time. However, the main drawback of them is that they experience excessively high performance overhead. For example, in [22] that proposes a software-based DIFT implementation, the overhead reaches up to about 40 times the original code execution time in the worst case. The performance overhead of DROP [23], which proposes a ROP detection scheme, ranges from 1.9X to 21X. MoCFI [39], which introduces a CFI checking technique on ARM-based mobile devices, shows the performance loss about 5X. Considering that these techniques are usually employed for runtime monitoring, the performance degradation is not acceptable to be deployed in real machines.

2.2.2 Hardware-based Monitoring with Invasive Modification

To address the shortcoming of software-based monitoring, some early hardware approaches [18, 19, 27, 28, 40, 20] tried to improve performance by inserting into the host processor core dedicated hardware modules that accelerate monitoring computations. The main advantage of these approaches is that they do not need to instrument the host code and thus they could bring the overhead down to under 5%. However, they have a disadvantage in that invasive modifications to the processor internal (e.g., registers and pipeline data paths) are required. For instance in [18], inside the core, they installed hardware tagging units to conduct DIFT, called the *flow tracker* and *tag checker*, and widened the widths of registers, internal datapaths and caches, to accommodate tag bits, all of which call for major changes of the processor internal. In fact, modern microprocessor development may take several years and hundreds of engineers from an initial design to production [31]. Therefore, the substantial costs of development to integrate the customized logic would hamper processor vendors to adopt them, unless the necessity is clearly established.

2.2.3 Hardware-based Monitoring with Minimal Modification

In an attempt to minimize the internal architecture changes, the researchers in [20, 21] suggested security monitoring solutions in the existing multi-core environment where one general-purpose core is devoted solely to run a *helper thread* that performs tag propagation for the main code running concurrently on a different core. In [31, 41], they proposed an external device that performs monitoring outside the host. By dedicating the monitoring task to a separate core or an external hardware, these approaches can manage to enhance the performance drastically. However, as discussed earlier, the fundamental problem of these approaches is that a vast amount of information must be continuously delivered to the external hardware for accurate monitoring operations [41]. To cope with this communication issue, they modified either the x86 architecture to supplement special hardware queues and new instructions [20, 21], or the CPU pipeline datapath to provide a customized channel between the host and the external device [31]. Our work is somewhat similar to the work in [31] since both propose the external hardware optimized for DIFT. But ours is different from theirs in that we exploit the standard interface CDI for communication. The security engines proposed in our work have been specially designed to perform the monitoring tasks by interpreting the signals for debugging from CDI.

2.2.4 Hardware-based Kernel Integrity Monitors

The idea of using a separate hardware module to secure the kernel was first proposed by Hollingworth et al [42]. In the work, they used a symmetric multi-processor to secure the kernel. Later, Zhang et al. [43] suggested deploying an intrusion detection system on a coprocessor to monitor the host. Petroni et al. designed and implemented Copilot [13], a snapshot-based kernel integrity monitor based on a coprocessor PCI card. Originally, Copilot focused on monitoring only *immutable kernel regions*, where the residing code or data should not be changed during runtime, but additional effort had been made in [25] to monitor *kernel dynamic regions*, where the data can be modified as a result of legitimate kernel operations, as well. Baliga et al. presented a PCI card-based two-phase rootkit detection technique in which the first phase automatically extracts the invariants of kernel data structures and the second phase enforces the invariants at run-time [24]. Even though these monitors employed external hardware module to secure the kerel, they employed snapshot-based approach, which periodically takes snapshot of memory contents to detect suspicious activities. Therefore, they cannot constantly monitor the memory contents since doing so would impose performance overhead on the system. Moreover, their periodic monitoring opens up a new vulnerability, which can be exploited by *transient attacks* as reported in [10].

Recently, researchers who are aware of the problem of transient attacks proposed snoop-based external monitors. Vigilare [10] was the first work that leverages an external hardware monitor to constantly snoop the system bus to protect immutable kernel regions. In [11], they also proposed snooping techniques. They especially has shown the feasibility of the approach by showing that only small portion of all memory access are related to operating system kernel and the portion of write in the access is again small. KI-Mon [12] extended the Vigilare to monitor both immutable and mutable kernel regions. These monitors have the capacity of defending a system with negligible performance overhead, but the scope of attacks they can address is somewhat limited in that they can monitor only memory access patterns placed on the standard bus interface.

2.2.5 Utilizing debug interface

The idea of using debug interface for other purpose rather than debugging has been proposed in other works, especially in the field of fault-tolerent computing [44, 45, 46, 47, 48, 49, 50]. In [44, 45, 46, 47], they proposed methods of injecting faults to the host system by accessing internal resources such as registers and memory via existing OCD. In doing so, they tried to facilitate fault tolerance evaluation. Their works are different from ours because they use OCD to intrusively modify the internal state of the host CPU to inject faults while ours are directly connected to CDI to monitor the trace of memory access events so as to detect malicious attempt. In [48, 49, 50], they presented error detection approaches utiilizing available debugging interface, CDI or trace buffer in OCD, to retrieve control flow information as well as load/store information. The overall concept of exploiting information flowing out of CDI is same to the of our monitoring system, but the main objective is different in that theirs is to detect faults while ours is to defend the host system against various security attacks.

Chapter 3

Monitoring the Integrity of OS Kernels with Data-Flow Information

3.1 Introduction

As electronic devices such as PCs and smartphones become essential parts of our everyday life, the potential privacy and security risks due to numerous *malwares* on the devices are rapidly growing. As a means to protect such devices from these attacks, current OSes support a variety of anti-malware solutions. These solutions usually depend on the services from the underlying OS kernel, implying that they would only work as designed when the integrity of the kernel is ensured. However, the kernel integrity has been seriously threatened since the advent of *kernel level rootkits* that manipulate the kernel so as to achieve certain goals (i.e., concealing their existence or providing backdoor accesses). Because the kernel operates at the highest privilege level in the system, the compromised kernel may nullify the effectiveness of any anti-malware measures that have their root of trust on the kernel.

The threat of rootkits have urged researchers to conduct much study to seek a more secure computing base that can safely monitor the system and ensure the kernel integrity even in the presence of rootkits. Two mainstream of the research directions are *hypervisor-based* [14, 8, 9] and *hardware-based* approaches [7, 13, 10, 12, 11]. In general, the former approaches have popularity in the security community as they do not necessitate underlying hardware modification while providing a higher privileged, thus safer, software layer for monitoring than the kernel does. However, the latest attacks [51] and reported vulnerabilities [52] pointed toward the probability that the code and data of hypervisors can also be compromised at runtime. Although the known vulnerabilities have been fixed shortly, the growing complexity of hypervisors implicates that there would be more vulnerabilities revealed in the near future.

The hardware-based approaches utilize an isolated hardware module physically independent of the monitored host system [13, 10, 12, 11]. In particular, prominent monitoring schemes are recently proposed in [10, 12, 11]. At the center of these approaches, there is a hardware monitor, which we hereafter call the *snoop-based monitor*, whose role is to detect malicious attempts to alter the kernel by snooping every data traffic between the host CPU and main memory. Being located at the outside of the host as a dedicated hardware unit, the monitor is not only immune to rootkits attacks on the host, but also able to constantly observe the memory access behaviors of rootkits revealed on the system bus without affecting the host performance.

Although snoop-based monitors have been working well in their environments and assumptions, we have recently discovered a potential vulnerability which future attackers might exploit. It comes from the fact that most computer systems employ write-back caches. Being located in between the host CPU and main memory, caches hold copies of data or instructions recently accessed by CPU, thereby boosting the overall system performance to a large extent. However, for the perspective of snoop-based monitors, the existence of caches can be disadvantageous because they shall reduce the number of events that the monitors can watch. For example, if a rootkit tries to compromise the kernel by modifying sensitive data, and the very data hits in the cache, then the write traffic would not appear on the system bus, rendering the monitor oblivious of the write event.

Even though some previous works discussed the possibility that this problem may seriously undermine the effectiveness of their approaches [10, 12, 11], none of them has properly addressed this *cache-induced hiding* (CIH) effect problem. In [12], they tried to avert the problem by restricting the usage of their monitors to the systems with write-through caches. In [11], they merely mentioned a simple scheme of using periodic cache flush. Unfortunately, they did not provide any empirical data about how much loss their scheme may suffer on performance, detection rate or power consumption. However, as we will see later, our study evinces that frequent cache flush might increase the host performance overhead to a large extent.

In this chapter, we present a hardware-assisted low-overhead solution which thwarts the CIH effect by enabling the external monitors to directly access the *cache resident information* (CRI) which includes all the internal data residing within the cache without being exposed on the system bus. To implement this solution, we utilized the existing hardware logic, CDI, which can be found in several processors available today such as ARM Cortex series and Xilinx MicroBlaze[35, 34]. If CDI is plugged into a security monitor, the bountiful information provided by CDI, which contains memory access events issued by CPU, would certainly help monitor perform its desired task without the CIH effect.

This task, however, involves several complications in implementation majorly because the initial set of signals from CDI cannot be simply fed into the security monitor as they are in their present form. Some signals originally generated for debugging must be translated into another form that is required for security monitoring. Therefore, we have developed an extra hardware unit, called the *Extrax*, that being located between CDI and security monitors, carefully examine and properly refine or transform each individual signal from the interface before delivering it to the monitor.

To validate our design and further explore the implication of this additional circuits to the overall system, we have implemented a full snoop-based monitoring system in which the host system has been augmented with Extrax. With the system prototyped on a FPGA platform, we evaluated and compared the performance, power and area of our full system against the baseline system in which Extrax is not deployed. Experiment results exhibit that our monitor, with modest area and power overhead but with the host performance being almost unaffected, successfully detects rootkit attacks regardless of the type of caches while the baseline monitor often fails.

The rest of the chapter is organized as follows. We first present a motivational example in Section 3.2. Then in Section 3.3, the assumption and threat model are presented. In Section 3.4, the baseline system is presented to show the overall operation of hardware-based monitoring. Section 3.5

describes the details of the proposed security extension, Extrax. After Section 3.6 shows our experimental results, we conclude this chapter in Section 3.8.

3.2 Motivational Example

We define *cache resident attacks* as malicious attacks that, intentionally or unintentionally take advantage of the CIH effect; the existence of write-back caches can unintentionally blindfold snoop-based monitors by impeding memory write events from appearing on the system bus, or attackers can intentionally hide the evidence of attacks by overwriting the malicious data residing in caches with benign one, thereby prohibiting the monitors to detect the symptom of attacks. To better explain, we chose the *loadable kernel module (LKM) hiding* technique as a representative cache resident attack example since many rootkits in the wild employ the technique to hide themselves. LKMs are initially designed to support extension of the kernel code at runtime without recompiling the entire kernel. However, they are often used by attackers to conceal malicious processes, files or even themselves from detection mechanisms. Adversaries achieve their goal of hiding LKMs by directly modifying the kernel data structures that maintain the list of loaded LKMs.

Figure 1 shows how the LKM hiding technique is affected by write-back caches. In (a), there are several LKMs, each of which is represented by the **struct module**. The kernel handles the LKMs by maintaining the **modules** list, which is a linked list of **struct module**. Upon the module load request, in this case a request from the malicious LKM depicted in (b), the kernel

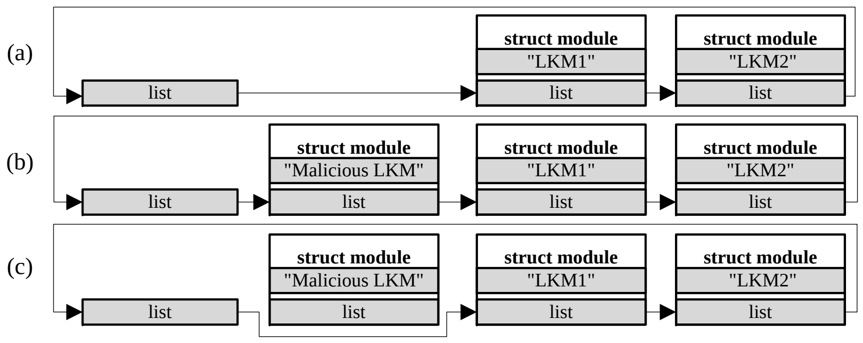


Figure 1: Cache resident LKM hiding attack

adds the corresponding **struct module** to the **list**, which is the head of the **modules** list. In a system with write-back cache, the **list** will be cached after this step, and subsequent accesses to the data structure will also hit in the cache. Thus, even if the malicious LKM removes itself from the **modules** list by directly manipulating the pointers of the **modules** list as depicted in (c), this event might not be placed on the system bus. Consequently, recently proposed snoop-based monitors might no longer guarantee the integrity of the kernel since they detect attacks by snooping the system bus. Hence, a novel way to nullify CIH effect should be devised.

3.3 Assumptions and Threat Models

We use the assumption taken by previous snoop-based monitors, especially by KI-Mon [12]. Therefore, we assume that adversaries have already gained administrators' privilege on the host system and thus are able to install rootkits to hide themselves or leave backdoors to the host system; for instance, the attackers can install LKMs or place hooks on critical system calls. However, we rule out *physical attacks* by an insider who has direct access to the host system and *direct kernel structure manipulation attacks* proposed in [53].

In addition, we also assume that the host system uses write-back caches, and provides CDI, that can be connected to OCD. Side-channel attacks that exploit the information from OCD/CDI are not considered in this work.

3.4 The Baseline System

3.4.1 The Overall System Design

Figure 2 shows a high level view of the baseline system. Since our monitoring system employs the snoop-based monitoring scheme, it is similar to the prototype of KI-Mon[12]. To ensure the integrity of the kernel, the monitor side core dynamically configures the hardware ASIC units, especially the *snooper*, based on a security policy.

We designed our baseline monitor to support various policies on detecting attacks on immutable regions and *mutable objects* that have *invariant value sets*, as KI-Mon proposed in [12]. Immutable regions contain data that

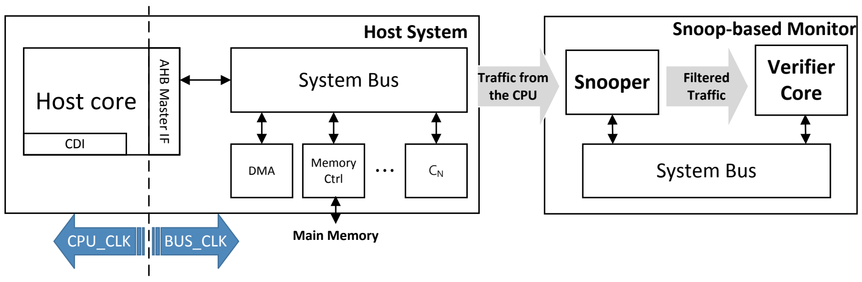


Figure 2: The overall baseline system design

should not be modified after the boot process is complete, such as the *system call table* (SCT) and *interrupt descriptor table* (IDT). Kernel mutable object with a invariant value set is data object in which data can be updated by the kernel at run-time, but the updated value is chosen among the set of possible values that can be profiled prior to run-time; for instance, many function pointers within kernel objects are known that each function pointer points to one of its possible candidate landing sites [12].

According to the security policy of the host CPU, the snooper is configured with appropriate address ranges of the kernel data objects or regions to be monitored on main memory. Then the snooper constantly acquires write events placed on the system bus, filters out every benign event that is not relevant to the monitored regions, and transfers only the ones that violate the current security policy to the *verifier core* for further investigation.

While attacks on immutable regions can be easily detected by simply snooping the bus for write events on the region, catching evidence of malicious modifications on kernel mutable objects with invariant value sets is not straightforward since mere write events cannot be regarded as a symptom of attacks. Therefore, we employ techniques similar to the ones proposed in [12] such as the *whitelisting-based verification* and *callback-based semantic verification* that basically verifies the written values as well. The detailed explanation of these techniques are omitted in this chapter since our work is focused on overcoming the CIH effect rather than suggesting new detection schemes. Therefore, readers interested in these techniques are kindly referred to [12].

The key difference between our baseline system and the prototype of

[12] is that ours uses write-back caches. Therefore, snooping only the system bus may cause detection failure because of the aforementioned CIH effect.

3.4.2 Periodic Cache Flush for Cache Resident Attacks

As mentioned in [11], periodically flushing caches might help reveal more CRI on the bus when write-back caches are deployed in the system. To show the effectiveness of the scheme, we applied it to the baseline system. When implementing the scheme, the flush period should be decided with great care because a reckless choice of the period may induce either non-negligible overhead or detection failure.

For attacks that aim at immutable regions [10], the cache flush period, **p**, can be selected arbitrarily because any write attempt on the regions is deemed malicious [10], and the cacheline where the written data is located will eventually be evicted to main memory regardless of the period **p**. However, the decision of the period **p** for attacks that target kernel mutable objects is far more difficult since attackers can usually figure out a way to avoid detection by slightly modifying the original attacks.

To better explain, consider the case shown in Figure 1. Assume that the state of the **modules** list changes from the state (a) to (b) on time **s**, and from the state (b) to (c) on time **e**. In principle, the baseline monitor concludes that an LKM is malicious when the LKM is removed from the **modules** list (state change from (b) to (c)) while the corresponding memory region for the LKM remains in memory [12]. Thus, the period of cache flush **p** should be shorter than the interval **d=e-s** so that every event on **modules** list can be revealed on the system bus before the adversary achieve her own goal. Since **d** in a primitive LKM hiding technique is sufficiently large, **p** of our baseline with periodic cache flush could be long as well, so as to detect the attack with acceptable performance overhead.

However, by slightly changing the original LKM hiding technique, it is possible for attackers to reduce **d** substantially. The devised technique requires two LKMs, one of which is the malicious LKM and the other is an LKM that merely hides the first one. We call the latter the *hider LKM* whose role is to insert a callback function to the kernel timer, and set the timer with a period **q**. Then the callback function is periodically invoked to check whether the malicious LKM, which ultimately achieves the attacker¡¯s goal, is inserted or not, and hide the newly added one upon detection. To insert and hide a malicious LKM, attackers first insert the hider LKM with an arbitrary period **q**, and insert the malicious one sometime later. Since the hider LKM does not hide itself and is thus added and removed legitimately, the monitor has no way of distinguishing the hider LKM from other normal LKMs.

Thus, to defend against such cache resident attacks, the flushing period **d** should be adjusted to a very small value. Our preliminary study showed that, in order to attain 100% detection rate, the period **d** need to be reduced to 30us, resulting in increased performance overhead of up to 84%. From this result, we claim that the detection with periodic cache flush might not only induce huge performace overhead, but would also cause failing in detection if attackers know the existence of periodic cache flush and modify their attacks to reduce **d**.

3.5 Extrax design

As long as CRI is accurately sent via CDI to snoop-based monitors, many security threats due to cache resident attacks would be resolved without modifying the host internal architecture. Unfortunately, realizing precise and efficient deliverance of these signals in an actual system comes at a cost with some implementation challenges, as briefly stated in Section 3.1. Below is summarized two of those that must be resolved in order to efficaciously transfer the host internal information to snoop-based monitors through the existing CDI.

1. CDI is originally designed to send a virtual address (VA) to the OCD unit for each memory access while the monitor demands physical addresses (PA) so as not to be disrupted by the certain type of attacks which will be discussed shortly. Therefore, the original VAs from CDI cannot be directly used for a snoop-based monitor.

2. The number of memory events coming from CDI is far larger than that of those appearing on the system bus because a majority of write events originating from CPU are hidden by caches on the way to the bus. This excessive number of events sent from CDI could be burdensome to the monitoring system in terms of performance.

These key issues are tackled respectively by two new hardware units, the *address translation unit* (ATU) and the *early stage filter* (ESF), both of which constitute our Extrax. Figure 3 depicts a block diagram of our proposed system where the baseline is extended with Extrax. It is noteworthy here that even though the information conveyed through ATU can cover all

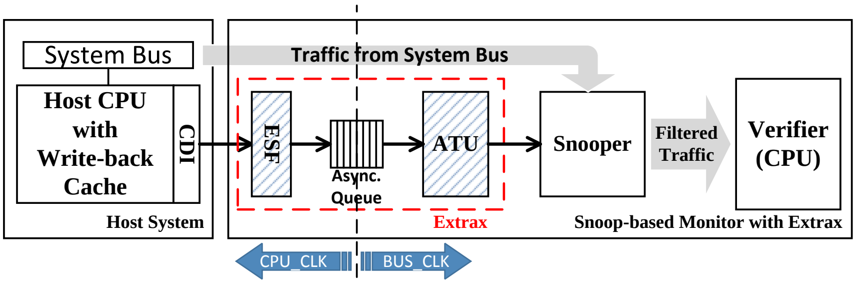


Figure 3: The augmented baseline system with Extrax

memory events, the original path from the system bus remains the same. The purpose of the path is mainly to detect attacks from other *bus masters* such as DMA. In this section, we will focus our discussion on these hardware units.

3.5.1 Address Translation Unit

Upon receiving VAs from CDI, a snoop-based monitor should decide whether the current memory access targets the regions it monitors. It seems that such a decision can be made based on VAs, but there are cases in which the monitor necessities PAs. For instance, consider the case depicted in Figure 4 where the monitor is protecting a kernel data structure contained in the critical page frame. Since this data structure is critical to the integrity of the system and is thus managed by the kernel through the *kernel page table*, no arbitrary mapping should be made to the data structure. However, in the presence of kernel-level rootkits, it would be possible for attackers to simply insert an LKM that generates another page table mapping for the data structure, denoted as the *malicious mapping* in the figure. Thus in this situation, if the monitor used the original VA to protect this type of kernel structures, the attackers could indirectly modify the kernel data with the

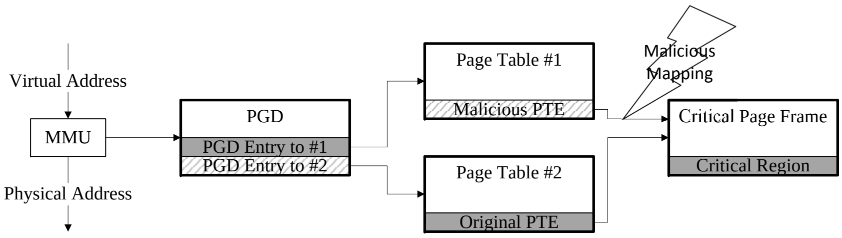


Figure 4: Multiple virtual addresses mapping example

newly mapped VA, hence successfully escaping the vigilance of the monitor.

To deal with the cases where PAs must be supplied for security monitors, we have installed ATU that translates VAs from CDI into physical ones. The overall architecture of ATU is displayed in Figure 5. ATU is configured by the monitor through the advanced high-performance bus (AHB) slave interface. On a translation lookaside buffer (TLB) miss, a *page table walk* is initiated through the AHB master interface, which is connected to the host system bus. The input to ATU includes a VA, the *context ID* and the base address of the Linux page global directory (PGD). The former two inputs are provided by CDI, while the latter one is configured through the AHB master interface immediately after the current context ID is updated. TLB is a fully

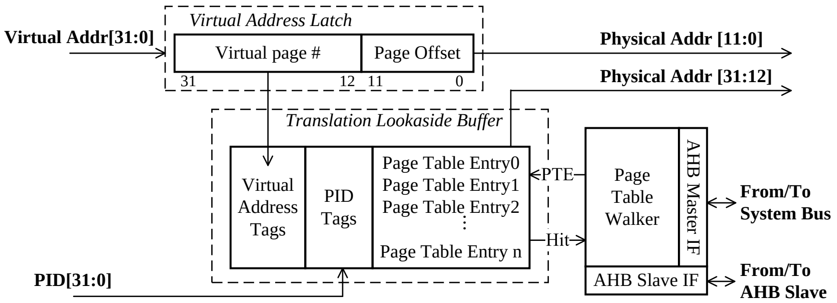


Figure 5: The structure of the ATU

associative table in which the number of entries can be configured from 16 to 32. It has a random replacement policy.

3.5.2 Early Stage Filter

With the help of CDI, our monitor is now able to snoop every write event generated by the host CPU without suffering from the CIH effect. This abundant information continuously streaming into the monitor will certainly enhance the chance to detect cache resident attacks. However, it may also create an excessively large volume of information flow that will inevitably impose heavy burdens on the monitor.

Though, if we remind that the monitors usually need to watch only a subset of the memory events of the host depending on security policies, it would be wasteful if ATU exhaustively translates all incoming addresses for the monitor. The problem can be alleviated if we can filter out benign events based on a security policy. As an example, for one of the policies considered in our experiments, the monitor is interested in write attempts to the kernel data. Therefore, any read memory events can be safely discarded before reaching either the monitor or even ATU. The filter operations in this case is in fact rather straightforward since memory access types are easily discernable right after the events occur. However, we often need to apply more aggressive filtering to the events. As briefly mentioned in Section 3.4, the kernel data of interest occupy relatively small amount of memory compared to the whole range of main memory. Therefore, the monitor just needs to watch the access events on this limited region. Unfortunately, it is not always straightforward to decide whether or not an event just emitting from

CDI falls into this region, since its address remains virtual before reaching ATU.

Nevertheless, there is still a way to provide a solution to this decision problem. For this, consider Figure 5 where we see that an address translation step does not require the *page offset* field; in fact, this field is identical for both VAs and PAs. Inspired by this fact, we implemented ESF which, being placed between CDI and ATU, removes unnecessary memory events based on page offset before the events reach ATU, thereby reducing the number of events delivered to ATU.

Figure 6 represents the internal block diagram of ESF, which rearranges the signals from CDI and filters out as many events as possible. The implementation of output rearrangement is somewhat simple in that it merely reorganizes and extracts signals that are needed for the monitors to detect attacks. Among the signals introduced in Section 2.1.1, the addresses/values of memory write instructions and context IDs are selected and rearranged for monitors.

In the current implementation, ESF has eight 12-bit *address range register* pairs and comparators. The address range register pairs of ESF contain the

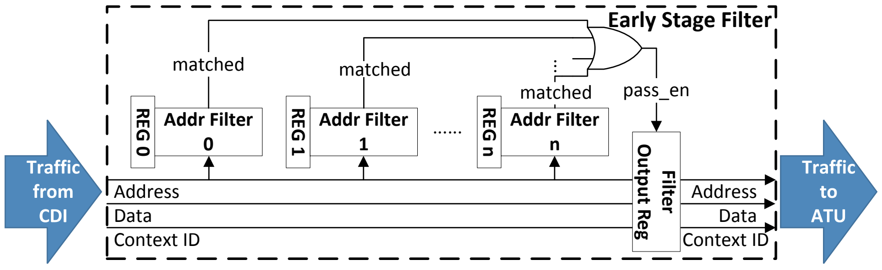


Figure 6: The overall structure of the early stage filter

page offset field of start/end addresses that need to be monitored. The register values can be configured by the verifier core at any time. Therefore, whenever the critical kernel regions of interest are updated, the monitor configures the snooper and ESF simultaneously.

After the configuration, ESF compare the page offset field of a VA that comes from CDI with the values of address range register pairs. If the address is included in any of the monitoring regions, ESF enables the filter output register so that the event can flow to ATU. Otherwise, ESF blocks the address, thus obviating unnecessary operations in ATU.

As mentioned before, current ESF implementation has 8 pairs of address range register, limiting the number of concurrent monitoring regions. Note that the number of registers can be adjusted for the environment of deployment. Alternatively, we can loosely set the address range register pairs, so that each pair contains more than one monitored region. Although it would produce unnecessary events for ATU to handle, ESF still do not miss any access to the monitored regions.

3.6 Experimental Results

3.6.1 Prototype System

We have implemented our baseline system as close as possible to the design proposed in [12], as an FPGA prototype where the host CPU is the SPARC V8 processor, a 32-bit synthesizable core [54] which uses a single-issue, in-order, 7-stage pipeline. It has separate 16kB L1 caches for instruction and data. In addition, the host CPU also has a 256kB L2 cache which employs the write-back policy. The host system bus compliant with the AMBA2 AHB/APB protocol is used to interconnect all modules in the system, and Linux 2.6.21.1 is used as the host OS. The snoop-based monitor system is also implemented with the same processor and the system bus. The snooper has eight sets of address range registers which can be configured according to the security policy of the monitor.

To evaluate our approach, we augmented the baseline system with Extrax. Although our host processor, open-source synthesizable core [54], provides their own CDI specification, the information comes out of CDI is quite restricted compared to that of commercial product, such as ARM. Therefore, we slightly extended it to support the CDI signals equivalent to those of ARM architecture (see Table 1). Thus, both ESF and ATU are implemented to be compatible with ARM CDI specification [55]). Since CDI is connected to both Extrax and OCD, we designed Extrax to disable signals to OCD when snoop-based monitoring is turned on. ESF is configured to have 8 address range register pairs. Our ATU, compliant with SPARC V8 Reference MMU [56], has been configured to have 16 TLB entries and 16 input queue entries.

Based on the parameters for the prototype as described above, we synthesized our system onto a prototyping board with a Xilinx SC5VLX330 FPGA. Table 2 provides the area of the baseline system and Extrax in terms of lookup tables for logic (LUTs), block RAMs (BRAMs) and DSP slices (DSP48E). It shows that Extrax incurs 12.09% overhead for LUTs as compared to the baseline hardware. Even though the area overhead of our Extrax seems non-negligible, it is noteworthy here that our baseline system, the

Category	Component	LUTs	BRAMs	DSP48E
Baseline System	SPARC V8 Core with L1/L2 Cache (Host System)	6856	86	4
	SPARC V8 Core with L1 Cache Only (External Monitor)	5878	15	4
	Bus components (AHB Buses + AHB/APB bridges)	908	0	0
	Memory Controller	57	0	0
	Snooper	3318	0	1
	Peripherals (TIMER, UART, and etc.)	2480	4	2
	Total Baseline System	**19497**	**105**	**11**
Extrax	Early Stage Filter (ESF)	502	0	0
	Address Translator Unit (ATU) including queue	1855	0	0
	Total Extrax	**2357**	**0**	**0**
	% Extrax over Baseline System	**12.09%**	**0.00%**	**0.00%**

table 2: Synthesis result of the prototype system

open-source synthesizable core based on SPARC V8 architecture [54], has indeed very small size. Therefore, we claim that the area overhead of Extrax might be quite acceptable if deployed on the system with commercial CPU core such as Cortex-A9.

3.6.2 Security Evaluation

To evaluate the security monitoring capability of our approach, we chose several well-known attack techniques that are employed in real-world rootkits [11] and implemented four rootkits that target either immutable regions or kernel mutable objects. Table 3 lists the rootkits, of which the specific target can be deduced by their names. The first two target immutable regions while the others, the LKM and *virtual file system* (VFS) hooking attacks

Example Name		BaseWT	BaseWB	Ours-Extrax
Immutable regions	**IDT Hooking**	Detected	Detected	Detected
	SCT Hooking	Detected	Detected	Detected
Mutable objects	**LKM Hiding**	Detected	Not detected	Detected
	VFS Hooking	Detected	Not detected	Detected

table 3: Rootkit detection result

target mutable objects that have invariant value sets. We also implemented monitoring software that runs on our verifier core to configure the peripheral units for monitoring, such as the snooper, ESF and ATU. The current address range registers of ESF is configured to capture memory accesses only on kernel mutable objects and other related data structures such as page tables for the objects. Memory events on immutable regions can be safely filtered out by ESF since, as mentioned before, snooping the system bus would be enough to catch attacks on immutable regions.

To demonstrate the effectiveness of Extrax, we injected the four rootkits into three system versions: (*BaseWT*) the baseline system with write-through caches as in [12], (*BaseWB*) the baseline system with write-back caches and (*Ours-Extrax*) our proposed system with Extrax. As seen in Table 3, the monitor in BaseWT could immediately detect all the rootkits since the host uses a write-through cache, thus immediately sending every write event onto the system bus. The monitor in BaseWB, however, was unable to detect attacks on kernel mutable objects because of the CIH effect. Ours-Extrax, on the contrary, could detect all the attacks (whether cache resident or not), thanks to our Extrax and CDI support.

Recent attackers tend to avoid launching attacks that are easily detectable like those on immutable regions. Instead, of more importance becomes the detection of attacks on mutable regions [25]. We have just seen that a snoop-based monitor deployed on the host core with a write-back cache is easily nullified when cache resident attacks are made on mutable objects. Therefore, we claim that Extrax can play a critical role in assisting such monitors, thereby increasing the security level of the systems.

3.6.3 Performance Analysis

Since CDI does not introduce any performance impact on the host, the main factor which incurs the overhead is the traffic generated by ATU as a result of address translation. To measure the performance overhead, we chose seven applications from the *SPEC 2006 benchmark suites* [57], and implemented two versions of the host system: the baseline and the proposed full system with ESF turned off. The reason of turning ESF off is to strain the system with the excessive traffic generated by CDI.

Table 4 presents this worst-case performance overhead, which is around 3.24%. The reason for this low overhead might be explained in a way that even if there seem to be a number of memory events coming from CDI, the number of events that really need memory translation in ATU is relatively small because our ATU has TLB and most memory translation end up retrieving values from the TLB. We also conducted the same experiment, with ESF turned on, monitoring the mutable objects related to the VFS hooking and LKM hiding attacks. As seen in the table, there is virtually no overhead caused by Extrax because ESF filters out most memory events that do not access the monitored memory regions.

Since turning off ESF does not cause serious performance overhead of 3.24%, some might think that ESF is not essential. As explained before, however, its main goal is to reduce the amount of CRI delivered to ATU. Reduced number of memory events would not only help ATU decrease the number of events that need address translation (main memory access), but it would also drastically reduces the number of TLB accesses, which in turn

Application	Baseline System	Proposed System with ESF turned-off	Proposed System with ESF turned-on
h264	10.46s	10.57s	10.46s
bzip2	788.05s	813.62s	788.05s
hmmer	39.96s	39.96s	39.96s
libquantum	10.11s	10.11s	10.11s
parser	1.41s	1.41s	1.41s
omnetpp	969.1s	975.35s	969.1s
xalan	3.43s	3.46s	3.43s

table 4: Performance overhead

might possibly save hugh amount of power consumed by ATU.

As a complementary experiment, the bandwidth of Extrax is shown in Table 5. This result is acquired by running the *STREAM* benchmarks [58] on our proposed system in order to obtain the worst-case bandwidth that can be produced by ATU, (meaning that no TLB hit occurs) meaning that it constantly requires page table walk and generates memory traffic on the system bus. Even in this unrealistically extreme case, the traffic placed on the system bus by ATU is relatively small compared to that of other components widely used and attached to the system bus of modern SoCs, indicating that the performance impact on the host system caused by Extrax should not be a serious concern.

Component	Bandwidth
USB 2.0 HS (High-Speed)	~ 480Mb/s
H.264 Codec (Full HD, 16-bpp, 60fps)	$\geq$ 1.85Gb/s
Camera Input Processing (Full HD, 24-bpp, 60fps	$\geq$ 2.78Gb/s
CPU Security Extension (Running at 200MHz	~ 275Mb/s

table 5: Bandwidth comparison

3.6.4 Power Consumption

To assess the power consumption of Extrax, we used Synopsys Design Compiler, Mentor Graphics ModelSim, and synthesized netlists of Snooper, ATU and ESF. Switching activity interchange format (SAIF) files were extracted from Modelsim with synthesized input test sequences that maximizes the power consumption of each component. Then the SAIF files and netlists were given as the input to Synopsys Design Compiler with a commercial 45 nm process library to estimate power consumption. The results are presented in Table 6 with other commodity processors as reference machines. As shown in the figure, Extrax consumes relatively small power as being compared to commodity processor cores in products ranging from low- to high-end computing devices.

3.7 Limitation and Future Work

Our monitoring system could overcome the problem of previous snoop-based monitors by employing hardware security extension, Extrax, that provides the external monitor with CRI. Even though our scheme has made a progress in raising the security level that snoop-based monitors can provide, the snoop-based monitoring is still not at its full maturity. In this section, we

Component	**ARM Cortex-A9 (Dual Core, no L2)**	**ARM Cortex-A15 (Dual Core, 1MB L2)**	**SPARC V8 (Single Core, 256KB L2)**	**Snooper**	**ATU**	**ESF**
Process	40nm	30nm	45nm	45nm	45nm	45nm
Power Consumption	0.5W (@ 800 MHz) /1.9W (@ 2GHz)	3W (@ 1.7GHz)	188mW (@200 MHz)	7.24mW (@200MHz)	4.84mW (@200MHz)	560.8uW (@200MHz)

table 6: Power consumption analysis

describe the limitations of our monitoring system.

Our prototype system is currently equipped with a total of 8 pairs of configuration registers, in both Snooper and ESF, to store the start/end addresses of monitored regions. Although we can increase the exact number of registers for the systems that allows it, we cannot say that it is always possible to store all the address pairs. In such a case, we can set a pair of registers to represent a memory region that include more than one regions that we have to watch. This allows us to monitor more than one regions with a pair of registers, since we still do not miss any access to the memory regions. Since this loosely set monitoring region would produce unnecessary memory events, Monitor software should perform extra work to drop these events. Our future work includes the investigation of trade-offs between the number of registers, hardware size, and the performance overhead caused by additional workload of Monitor Processor.

Our current system does not assume a type of attacks that target page tables, such that a rootkit manipulates the page table to allocate new physical pages, copy over the content of the physical pages that are being monitored, and change the mapping to the new physical pages. However, we claim that Extrax can detect them with slight modification that enables monitoring both virtual and physical addresses simultaneously. Even though the attack mentioned above changes the content of page table so that the virtual address can point to the different physical address, it still uses the same virtual address. Therefore, the augmented Extrax will be able to detect such type of attacks.

In this work, DoS attack by attackers who are aware of the existence and internals of Extrax is not considered. Such DoS attacks might be possible if attackers delicately craft their attacks to satisfy the following criteria. First, attacks should consist of consecutive memory writes. Second, the target addresses of the events should have the same 12-bit page-offset to that of the address monitored by ESF in order for the events to be delivered to the queue placed in between ESF and ATU. Third, the remaining 20-bit in each target address is different from each other to avoid TLB hit. Since the current Extrax is designed to drop events when the queue is full (even though, with the current setting, the queue never became full in our experiment), attackers might be able to conceal their trail by placing malicious write events among the benign consecutive write events that satisfy the above criteria. However, we claim that detection of such attack is not impossible because, as described above, the memory access behavior of the attack is quite restricted and have certain patterns. Therefore, Extrax could be augmented to detect this type of patterns and notify the monitor so as to log the incident for later inspection.

Our system does not assume attacks that Bahram et. al proposed in [53]. In their work, the attack named DKSM is introduced to show the vulnerability of virtual machine introspection tools. Since the attacks basically exploit the semantic gap between the external monitor and the host system, our monitoring system is also vulnerable to such attacks. However, such a vulnerability is not our monitoring system‘s own weakness, but is an innate weakness of all external monitors. One possible way to overcome the issue is to employ in-host agent [53, 59] that can deliver useful information to external monitors to bridge the semantic gap.

In the current work, we do not consider attacks that are performed by tampering with only processor registers. Even though devising such attacks, which leave the evidence only in registers but not in memory, caches or system bus, might seem to be quite difficult, it is theoritically possible. The most conceivable attack is to change the content of a special register such as TTBR of ARM or CR3 of x86 architecture [60, 61] to modify the base address of *page global directory (PGD)*. If such an attack is successfully launched, our monitor has no way to figure out the exact location of the page tables that the processes is really using. Since ATU of Extrax rely on the integrity of the exact location of the page tables, the attack would nullify Extrax. Our future works include a development of a mechanism that can safely deliver the content in registers to the outside the host.

Even though our current prototype is implemented with a rather old-fashioned processor and old linux version, but the implementation is not restricted to the technology or specific linux version because the purpose of our work is to show readers the proof of concept of our system. Currently, we are migrating this prototype system to the platform on which ARM Cortex series is used as the host processor.

3.8 Conclusion

In this work, we proposed to reuse the CDI feature readily available for debugging in modern CPU cores in an effort to elevate the effectiveness of existing snoop-based monitors. We first discussed several implementation complications involved in the transfer of CDI signals for snoop-based monitors located outside the host CPU. Then we suggested Extrax which is an ASIC module plugged into CDI in order to convey the host internal information from CDI to the external monitor. For precise and efficient monitoring, the module performs the tasks of address translations and filtering out benign memory write events. To validate our proposed design, we have implemented a prototype on FPGA, and evaluated the security capabilities in addition to the performance, power and area overhead. Empirical results showed that our monitor, regardless of the type of caches, successfully detect all our rootkit samples, which the previous monitoring systems often failed to catch owing to CIH effect, with modest area and power overhead increase along with virtually no host performance overhead.

Chapter 4

Monitoring Dynamic Information Flow using Control-Flow/Data-Flow Information

4.1 Introduction

DIFT detects a variety of malicious system behaviors that intend to compromise computer systems or leak sensitive information [22]. Generally, DIFT sets up rules to tag (or taint) internal data of interest and keeps track of the taintness of their tags throughout the system [31]. At run time, every data derived from the one with tainted tag has its tag tainted. An alarm will be triggered as soon as any of the tainted data involves in potentially illegal activities, such as pointing inside the prohibited code or being included in a data stream on the output channels. DIFT does not depend on static patterns or signatures of attackers but on their dynamic behaviors at run time. So, it is effective to defend against new attacks whose patterns are not known yet, and to block any unsafe operations on sensitive data even if the data is encrypted [16].

DIFT has been implemented in various forms of either software or hardware. Most software approaches add instrumented code into the original application to track the propagation of tainted data [16, 22]. The key advantage is that they can perform DIFT simply by programming their algorithms. Not surprisingly, however, they show too large computing overhead to be deployed in practice. Even after much effort [16, 17], the overhead still remains one or two orders of magnitude higher than that of hardware approaches [18, 19] in which extra hardware for DIFT operations is designed and integrated into an existing processor for acceleration. The hardware typically consists of logic blocks that monitor the execution of each instruction in the processor and keep track of tag information flowing from the execution unit at every cycle.

Unfortunately, the remarkable speed of hardware DIFT comes at a cost. To maximize the performance, the hardware has been tightly integrated inside the processor. However, such integration mandates major modifications to processor internal components such as registers and pipeline datapaths, thus substantially increasing the time and cost for re-manufacturing existing processor core architecture [31]. As alternatives to mitigate this problem, there have been more recent studies [31, 20, 21] that propose the techniques aiming to minimize the change to the processor core internal. In their approaches, the host processor can concentrate on the execution of its own code while the time-consuming tag propagation work for DIFT is offloaded to the DIFT hardware device outside the processor. In the literature, they empirically demonstrated that DIFT can be carried out in a great speed by external hardware, relieving significant burden for DIFT computation from the host. However, there still remains a great challenge to overcome for the success of these approaches. The challenge originates from the limited ability of an external device to monitor every internal state change dynamically made by the code running on the host. For precise DIFT, the external monitor should be able to receive from the host virtually all essential runtime information including branch targets, memory addresses and register moves, which will incur a tremendous amount of traffic for communication between the two devices. In [21, 20], they report that the communication overhead may account for up to 30% of the total execution time even after all their optimizations through hardware communication buffers and special instructions. In [31], this overhead issue was treated more aggressively by modifying the host architecture in a way that a customized interface can be embedded into the processor pipelines. Through this interface, their external device was able to have a special connection for extracting any runtime information for DIFT computation directly from the internal pipelines with very little overhead.

In this chapter, we introduce our recent work on building a hardware DIFT engine. Our approach is similar to those in [21, 20, 31] in that our engine is also connected externally to the host processor. But looking at the details, ours is different from them in several aspects. One main difference is that our approach does not modify internally the host architecture to provide a DIFT-customized interface or connection for the external engine. In our system, the engine is connected to the processor via CDI.

Being plugged into CDI, our DIFT engine has full access to the bountiful information transmitted from CDI. However, as already explained in the previous chapter, the set of CDI signals cannot be simply fed into the DIFT engine, and they must be refined and filtered into what are suitable for DIFT computation. Therefore in our design, between the DIFT engine and CDI, there lies a component, called the *CDI filter*, which, taking the CDI signals as input, filters the signals properly before delivering them to the engine. In Section 4.2, we characterize DIFT computations, and discuss how DIFT works on our computing system with an external engine for DIFT. Then in Section 4.3, we describe in detail the hardware structure of our DIFT engine, and explain how the engine efficiently receives all necessary runtime information through the CDI filter. Experimental results in Section 4.4 show that our engine successfully operates at extremely high speed to provide ample protection against various attacks.

4.2 DIFT Process with an External Hardware Engine

DIFT has been applied to analyze the runtime behaviors of diverse types of attacks, such as *SQL injections*, *buffer overflows* and *data leak prevention* (DLP). In this section, as an example, we explain the DIFT process to guarantee DLP and how it works in our computing framework for DIFT where a DIFT hardware engine is connected to the host processor.

Generally, the first step of DIFT for DLP is *tag initialization* where

Attacker Code for Data Leak	**DIFT for DLP**
1. file_ptr = file_open("Password"); 2. data = read (file_ptr); 3. encrypted_data = encryption (data); 4. data_leak (encrypted_data);	1. tag [file_ptr] = "sensitive" 2. tag [data] = tag [file_ptr]; 3. tag [encrypted_data] = tag [data]; 4. if (tag [encrypted_data] == "sensitive") Exception!!
(a)	(b)

Figure 7: Example for DLP using DIFT

the input data from confidential sources are tagged as *sensitive*. After tag initialization, follows the *tag propagation* step in which any new data derived from the tagged data is also tagged. Tag propagation continues through code execution. When there is any attempt to extract some data toward outer world, such as sending data over the network or saving it to a storage device, the data is checked whether it is tagged or not. If any tagged data is detected at the *tag check* step, a security exception will be raised. Figure 7 presents a code example to illustrate the DIFT process that ensures DLP. Lines 1 and 4 correspond respectively to the tag initialization and tag check stages, and lines 2 and 3 to the tag propagation stage. In our system, the host OS kernel takes responsibility of the first two stages, and our engine of the last one partially because tag propagation is the pivotal and most time-consuming task in DIFT. To denote the tagging in its tag propagation, our engine associates a tag bit with each data location such as registers and memory. When data is tagged, it taints the tag bit by setting the bit on.

We now explain how the attack in the example can be detected by DIFT whether or not the data is encrypted. First, for tag initialization, certain files are to be labeled as sensitive sources. In Figure 7, the file *Password* is assumed to be sensitive. In the left column, we see that as the first stage of attack, the adversary code obtains the file pointer after opening the sensitive file, and then reads sensitive data from the file. To detect this trial of attack, the kernel compares a file name to the *list of sensitive files* when the file gets open. To enable this, we have modified system calls for file accesses, such as **open**, so that the kernel can be aware of every access of an application to any file in the system. Since this step requires interaction between the host and the

	Original Code	**Tag Propagation**
1	ldr r9, [r0, #0x40]	tag[r9] = tag[r0] or tag[deref[r0]]
2	add r3, r9, r3	tag[r3] = tag[r9] or tag[r3]
3	orr r9, r9, #0xc0	tag[r9] = tag[r9]
4	sub r2, r9, #0xf	tag[r2] = tag[r9]
5	str r2, [r1]	tag[deref[r1]] = tag[r2]
	(a)	(b)

Figure 8: Example of tag propagation rules

DIFT engine, we have implemented a tag initialization function as a device driver that basically reports to the engine the location (i.e., register number or memory address) of the data that need to be tainted. Then the engine, in return, taints the associated tags for the location in the report delivered from the kernel. In Figure 7, the file is sensitive, and so the system call initiates a procedure in which the engine taints the tag of the file pointer by setting the tag bit on.

For tag propagation, any data read from the file is tainted to denote being sensitive because its file pointer tag is on. Even when the data is encrypted, it would be tainted because the outcome of an encrypt function should be tainted if the input is tainted. A tag is propagated in a machine instruction from a source operand to the destination operand based on a set of *tag propagation rules* which are specified at the granularity of basic operations such as arithmetic and logical operations. Figure 8 shows a segment of the host code and its associated propagation rules with operands. In the figure, the propagation rule at line 2 in (b) depicts that the **or** operation needs to be performed on the tags of **r9** and **r3** before the result is propagated to the tag of **r3**. In short, two propagation rules, **or** and **=**, are applied when the original code at line 2 is executed. From this example, we learn that for the generation of a tag propagation rule, the DIFT engine must decode the given instruction and identify its opcode and operands.

The tag propagation task on our engine is basically determining whether each tag should be on or off as the host code executes. At the beginning of the execution, the engine allocates one tag bit per CPU register and memory word. Every time the host executes an instruction, the engine also carries out the corresponding propagation rule like those shown in Figure 8. Since this rule is generated from each instruction at runtime, the engine first fetches the same instruction from the main memory that the host CPU just did, and tries to resolve the operand values in order to locate every tag operand for its tag operations. However, not all values can be resolved only by decoding the instruction. For instance in Figure 8, the load instruction at line 1 uses two operands: register and memory. For correct tag operations, the engine must have the exact register number and memory address. While the former is trivially found (i.e., **r9**) right from the instruction, the latter remains unknown since the value of **r0** is hidden inside the host CPU. In our system, therefore, such hidden information is forced to flow from the host into the DIFT engine in a stream of the data values which we call *runtime traces*.

At the last stage of attack, the data will be leaked through network. The operations in the right column display a sequence of DIFT actions each corresponding to a statement in the attacker code. At line 4, the data is about to be transferred outside through an output channel. For DLP, the kernel must check the tag of the data given to the channel. For this tag checking step, we installed a function into the system calls involved in data output channels, such as network packet generation. When data is to be carried outside in a network packet through an output channel, this kernel function checks the data tag with the assistance of our DIFT engine. As the first step of this check, the function makes an inquiry to the engine with the location of the data being transferred out. Upon receiving the inquiry, the engine checks the tag value, and notifies the host of the tag checking result. Once the kernel receives the result, it finally checks whether the data is leaked as part of legitimate operations (e.g., bank transactions) or not. If the tainted data is leaked as a result of unauthorized operations, the kernel raises an alarm. Note that deciding the legitimacy of certain operations is in fact beyond the scope of this work as it is irrelevant to the design of our DIFT engine.

4.3 Building a DIFT Engine for CDI

In this section, we describe how our DIFT engine is implemented to fully support the three stages of the DIFT process defined in Section 4.2.

4.3.1 Components of the DIFT Engine

The overall SoC design for our DIFT solution is presented in Figure 9 which shows the interconnections between the host CPU core and the DIFT engine. Within our SoC platform, the engine is connected via CDI and a generic shared interconnect to the host CPU along with other hardware modules. It has both the master and slave interfaces so that it cannot only respond to the interconnect transactions from other modules, but also initiate

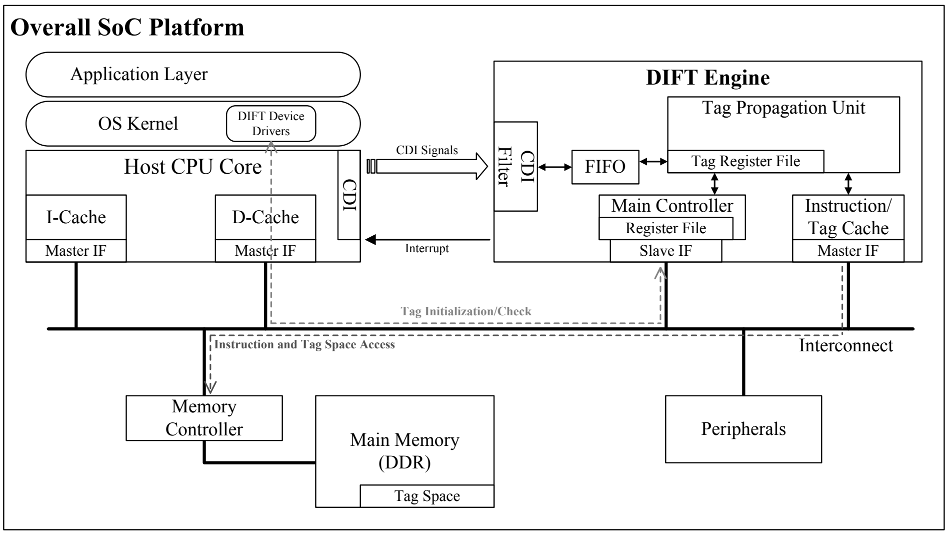


Figure 9: Overall SoC platform

transactions to access data in the modules. In addition, it can send an interrupt to the host whenever necessary, which would help to reduce the polling overhead incurred during communication between the host CPU and the DIFT engine.

The primary purpose of CDI is to efficiently support the communication for runtime traces flowing from the host to the DIFT engine. A simple way for this without CDI is to use a generic shared interconnect used as the system bus. Of course however, it would consume more bus cycles that normal data transactions could otherwise use. It should also spend extra CPU cycles in executing instructions for the delivery. In this work, to reduce these overheads, we have devised CDI to become a special channel for runtime traces such that a trace can be transmitted from the host into our engine consuming neither CPU nor bus cycles.

As explained earlier, CDI is a CPU side interface built in various modern processors, which helps users verify the functionality and/or analyze the performance of their applications. It is usually connected to the OCD unit that allows the users to watch the control paths that their target processor has taken during code execution, and to examine the values in registers and memory locations. Among the types of signals listed in Table 1 in Chapter 2, the DIFT engine does not need all these signals as the runtime traces. Thus, we implemented the CDI filter to drop useless signals before they reach the engine. In our current implementation, the traces emitting from the filter include the current process ID (PID), the address of memory data accessed by a load/store and the program counter (PC) value for the current instruction address. Another important role of the CDI filter is to infer the existence of a branch instruction in the current host execution path. In our implementation, we use a simple heuristic where an abrupt change of the PC values in runtime traces is a sign of the existence. This heuristic is based on a common observation that PC is normally incremented by the instruction word length without (un)conditional branches. As soon as the filter discovers such an abrupt change, it constructs a pair of addresses, the two PC values just before and after the change, each of which stands respectively for the branch address and the target address. These addresses are then delivered to the engine which uses them to grasp the host execution path from outside CPU.

The *main controller* in Figure 8 governs the communication between the host and the engine as well as all transactions related to DIFT computation. It is configured by the host to control the DIFT engine. By setting the values of the controller registers, the host can direct the operations of the engine, such as initialization and assignment of the functions for the *tag propagation unit* (TPU). As the central component of our engine, TPU processes all the tags that are associated with data storage in the host. Each entry in the *tag register file* (TRF) represents the tag for an individual register in the host CPU. Borrowing the idea from [40], the engine reserves a special region, called the *tag space*, in memory to stores a long array of bits each of which represents one word of host data in memory. To reduce the memory latency for accessing the tag space, TPU has a small cache, called the *tag cache* [31], for frequently accessed memory tags.

4.3.2 Tag Propagation Unit

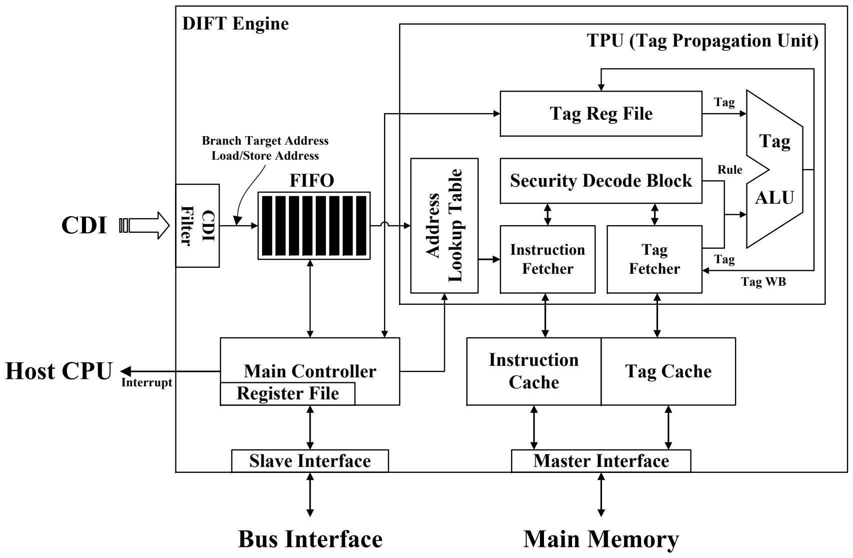


Figure 10: Microarchitecture of the proposed DIFT engine

Figure 10 represents the internal block diagram of our DIFT engine.

For unerring tag propagation, it is crucial that TPU correctly fetches host instructions from the main memory, following exactly the same execution path taken by the host CPU. Every fetched instruction enters the *security decode block* (SDB), which derives a tag propagation rule from the opcode and operands of the instruction, as shown in Section 4.2. If the operand is a register, TPU reads from TRF the tag register value corresponding to the operand. If the operand is the memory address for a load or store with register-indirect addressing (see Figure 8), TPU acquires the exact address from runtime traces by collecting each trace from the FIFO that temporarily holds all the traces out of the CDI filter. Then, it loads from the tag cache the tag bit representing the memory address. If a tag cache miss is taken place, the *tag fetcher* accesses the tag space allocated in memory to fill the requested line. Once all the tags are ready, TPU performs the tag propagation for the fetched instruction, and writes the result back to the tag bit for the destination operand in the instruction.

The idea of making TPU follow execution trails of the host brings about a couple of design challenges. One of them is that to follow the trails, TPU relies on the PC values carried in runtime traces, but the values are virtual addresses while TPU uses physical addresses to access the host memory. To resolve the discrepancy in these address spaces, we have the *address lookup table* (ALT) in TPU. An entry of ALT consists of the PID for a process running on the host and the virtual-to-physical address mapping information for the corresponding process. The mapping is determined by the host OS kernel when a new page is allocated for the code section of a process. Therefore, we have slightly modified several system calls related to page allocation in a way that whenever a page is allocated for a process, the mapping information along with its associated PID can be forwarded to TPU for ALT update. Fortunately for our design, a process usually holds only a few entries in ALT. This is because the code section ordinarily occupies a smaller number of pages than the data section. When a process is terminated, its entries are removed from ALT. For this procedure, we have also altered relevant system calls like **exit()**.

Another challenge here is that if TPU should always fetch instructions from memory, it could not catch up with the CPU speed certainly because memory is slow. To tackle this, we have the instruction cache (I-cache) in the DIFT engine. When TPU fetches an instruction, it first tries to load it from I-cache. If a miss occurs, TPU commands the *instruction fetcher* to read the entire cache line containing the instruction from the main memory. As soon as a branch is detected, TPU orders the fetcher to stop and wait until the branch result arrives from the host through runtime traces which continuously carry the PC values. If the branch is taken, the fetcher will load instructions from the address pointed by the new PC.

4.4 Experiment

To evaluate our approach, we have built a full-system FPGA prototype, where the host processor is the SPARC V8 processor, a 32-bit synthesizable core [62] which uses a single-issue, in-order, 7-stage pipeline. It has separate 4K-byte 2-way set associative instruction and data caches. The architecture of our DIFT engine has been implemented as described in Section 4.3.

Category	**Component**	**LUTs**	**BRAMs**
Baseline System	SPARC V8 Core (Host Processor)	4876	18
	Bus components	439	0
	Memory Controller	405	0
	Peripherals (TIMER, UART, and etc.)	963	2
	Total Baseline System	**6683**	**20**
DIFT Engine	Address Lookup Table	670	0
	AHB Master IF	154	0
	CDI Filter	27	0
	FIFO	129	0
	Instruction Cache	293	10
	Instruction/Tag Fetcher	97	0
	Main Controller	176	0
	Security Decode Block (SDB)	35	0
	Tag ALU/Tag Register File	109	0
	Tag Cache	180	2
	Total DIFT Engine	**1870**	**12**
	% DIFT Engine over Baseline System	**27.98%**	**60.00%**

table 7: Synthesis result

Even though our host core provides their own CDI specification [62], the information that comes out of CDI is quite restricted compared to that of commercial products, such as ARM. Thus, we slightly augmented our core to support the standard CDI signals that resemble those for the ETM of ARM [55]. We implemented the tag cache which is a 512-byte, 2-way set-associative cache with 4-byte cache lines, and the DIFT instruction cache which is a 4K-byte, 2-way set-associative cache with 32-byte cache lines. The bus compliant with AMBA2 AHB protocol [63] is used to interconnect all the modules in our prototype system. Linux 2.6.21.1 is used as our OS kernel and, as mentioned in the previous sections, it has been slightly modified to provide supports for our DIFT engine.

Based on the parameters for the prototype as described above, we synthesized our overall SoC Design onto the prototyping board with a Xilinx XC5VLX330 FPGA and 64MB external SDRAM. Table 7 provides the design statistics of our hardware prototype. We quantified the resources necessary for our DIFT engine in terms of lookup tables for logic (LUTs) and block RAMs. The design statistics shows that, compared to the baseline SPARC core, the DIFT engine incurs the resource overhead of 60.0% and 27.98% for BRAMs and LUTs, respectively. To complement the result, we also measured the gate count of the DIFT engine using Synopsys Design Compiler [64]. Synthesized with a commercial 45nm process library, our engine increases 11.98% of overall area over the baseline system. Although it may seem to be a large proportion, the actual gate-count of the engine is 212,051. Considering that recent computing platforms deploy more complex processors like ARM Cortex series compared to the one [62] we used in our experiment, we claim that the area overhead due to our security engine is acceptable in a more realistic hardware. (The gate-count of Cortex-A9 processor with 45 nm process is about 26 M [65].)

To estimate the power consumption of the monitoring engine, we simulated the engine using its synthesized netlists on Modelsim [66]. As a result of the netlist simulation, the switching activity interchange format (SAIF) file is generated. Using the file as an input vector, we run the power estimation tools in Design Compiler with the 45nm process library. The power consumption of the DIFT engine is estimated as 205.4 mW at 1GHz operating clock frequency.

4.4.1 Security Evaluation

To test the security capability of our DIFT engine, we have synthesized the malware that encrypts a sensitive file named as ”secret.txt” and passes it through the network. Our malware, which is similar to the *Dorifel* [67] malware in the wild, has the ability of evading an intrusion detection system or signature-based DLP solution by using the AES encryption algorithm. However, as planned, any attempt to access the file from the malware will be detected by our modified **open** system call in the Linux kernel. When being detected, the kernel invoke the tag initialization function to taint the tag of the file pointer and to configure TPU to be ready for tag propagation. The malware naively proceeds and encrypts the data without knowing the existence of TPU, while our DIFT engine keeps track of the information flow by propagating tags. When the malware tries to leak the derived data, it invokes the **send** system call to transmit a message through the network. Because the system call is also modified to call the tag checking function, the kernel receives tags of the data from TPU as explained in Section 3, and decides whether to allow transfer the data outside or not.

4.4.2 Performance Evaluation

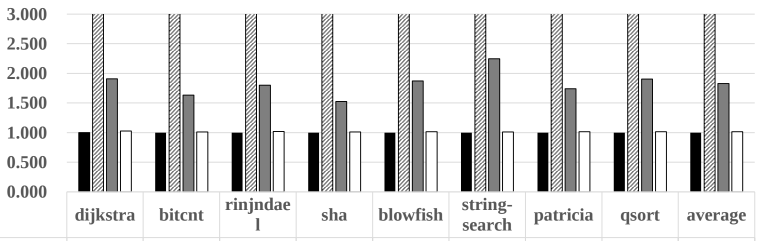
In order to measure the performance of our DIFT solution, we chose eight applications from the *mibench* benchmark suite [68]. The performance of our solution is compared with those of three systems that have different configurations. The first one, called *Native*, stands for a system that executes the original code with DIFT disabled. The *Software-only* solution employs a

	Native	Software-only	Software-DIFT	CDI-DIFT
dijkstra	1.000	17.785	1.909	1.028
bitcnt	1.000	9.298	1.631	1.011
rinjndael	1.000	47.193	1.799	1.018
sha	1.000	22.556	1.526	1.012
blowfish	1.000	47.147	1.873	1.015
string-search	1.000	17.102	2.247	1.012
patricia	1.000	16.269	1.740	1.016
qsort	1.000	10.503	1.905	1.015
average	1.000	23.482	1.829	1.016

table 8: Comparison table of execution time normalized to Native

software-instrumentation technique to augment the host code with instructions that perform DIFT computation on the host. *CDI-DIFT* refers to our DIFT solution that has an external DIFT engine connected to the host side CDI. In addition to these three systems, we added another configuration, named as *Software-DIFT*, that makes use of an external DIFT engine for time consuming tag propagation. The only difference compared to our solution is that the external DIFT engine does not have a connection via CDI to the host. Therefore, for the engine, the host must execute additional instructions to explicitly transmit runtime traces through the system bus. For this, we instrumented the host code with a set of instructions each of which is inserted after every branch and load/store instruction to send the updated traces to the DIFT engine. We used our in-house tool for code instrumentation.

In Table 11, we present the performance comparison of the four configurations. In the table, the host execution time of each configuration is normalized to that of *Native*. The results show that th *Software-only* solution suffers from an excessive performance overhead in that the total runtime is on average 23 times slower than that of *Native*. The overhead of *Software-DIFT* is less devastating than the *Software-only* version: it shows drastically reduced overhead of 82.9% as being compared to that of *Native*. However, it yet runs approximately 1.8 times slower than *Native*. The main cause of such tremendous overhead in both the configurations is the instructions added to the host code for delivering traces. On the other hand, *CDI-DIFT* substantially cuts the overhead down to 1.6% over *Native*. This amazing achievement is mainly due to the fact that, with the supplementary information coming out of CDI, no code instrumentation on the host is needed for our solution. The small amount of performance loss in *CDI-DIFT* is ascribed to the resource competition between the host processor and our DIFT engine because both are connected to the same interconnect and share the main memory.



	dijkstra	bitcnt	rinjndael	sha	blowfish	string-search	patricia	qsort	average
■ Native	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
▨ Software-only	17.785	9.298	47.193	22.556	47.147	17.102	16.269	10.503	23.482
◼ Software-DIFT	1.909	1.631	1.799	1.526	1.873	2.247	1.740	1.905	1.829
□ CDI-DIFT	1.028	1.011	1.018	1.012	1.015	1.012	1.016	1.015	1.016

Figure 11: Graph of execution time normalized to Native

4.5 Conclusion

This work presented a dedicated engine for DIFT. Our engine has been implemented and connected to the host processor interface via a standard bus interconnect so that no modifications are made to the processor internal. Nonetheless, being located outside the host system, the engine has limited visibility into the host internal states, which becomes a major stumbling block for successful DIFT computations on the engine. To overcome this limitation, we provide the engine with a separate communication channel through the existing debugging interface, called CDI, of the host. By receiving only the essential information filtered for DIFT out of the original CDI signals, the engine was able to perform its tag propagation task efficiently. Our experiments on FPGA prototype revealed that the engine successfully detects synthetic data leakage attacks with encryption, overcoming the limit of conventional DLP solutions. More importantly, our DIFT engine attains overwhelmingly low overhead, that is less than 2% for a group of mibench applications. The experiments also revealed that the area overhead of the hardware for our DIFT engine is acceptably small even when being compared to the normal sizes of today's mobile processors.

Chapter 5

Monitoring ROP/JOP Attacks using Control-Flow Information

5.1 Introduction

Code reuse attack (CRA) is a recently introduced technique that collects from the existing code blocks a set of small code sequences called *gadgets*, and chains them to perform malicious actions. Doing so empowers an adversary to perform Turing-complete computation without any attacker injected code [69], thus successfully defeating the well-known and widely adopted technique, generally called the W$\oplus$X (Write XOR eXecute) protection (or interchangably called the DEP) [28].

As the CRA threat is continuously escalating, many solutions have been proposed [70, 23, 27, 28, 29]. These solutions have come in various forms of either software or hardware. The clear advantage of software solutions is that they can be easily adapted to the present machine platform. Their drawback, however, is that they may impose tremendous computational loads upon the host machine mainly because the original program must be augmented with extra code that will be executed periodically to check abnormal control transfers on the host during runtime [70, 23]. On the other hand, hardware solutions [27, 28, 29, 30, 1] tend to exhibit high performance by accelerating the CRA detection process with the assistance of customized hardware logics for this task. In specific, authors in [27, 28, 29] proposed solutions where the hardware logics are tightly coupled with the host CPU for close monitoring of every control transfer during code execution.

Despite their dramatic performance enhancement, the main drawback of these approaches is that they require the redesign of the existing processor architecture, which would stymie the direct deployment of these solutions into commercial smart mobile devices. The reason is that such modification to the core internal is contradictory to the common design practice for a smart mobile device in industry today. As the central computing platform for applications running on the device, an *application processor* (AP) in the form of SoC lies in each device. To meet ever-increasing demands for low design cost, high performance and fast time-to-market, the general design rule of SoC is now to integrate commodity processors and supporting IPs (intellectual properties) for specific functions together. Thus, if the AP vendors adopt some of these hardware solutions for their products, they will be compelled to restructure the CPU core architectures, contrary to the general convention, thus resulting in tremendous cost for design and verification.

To facilitate the acceptance of hardware solutions for the CRA detection in today's smart mobile devices, some latest approaches [30, 1] endeavor to comply with the design rule of SoC. Their security hardware IPs are practical solutions for CRAs in a sense that they do not require any internal modifications to current host architectures but simply external connections with the host processor to build an SoC. The biggest challenge of the approaches however is that, being located outside the host processor, their hardware IPs are usually difficult to acquire the correct control flow information of the applications running inside the host, which is essential to monitor the existence of CRAs. In order to tackle this challenge, they exploit the built-in debug features to reveal the runtime information of the host to the outside of the core. Especially, the work by Lee et al. [1] implements a CRA monitoring hardware using the debug features supported in commercial ARM processors, which are the de-facto standard CPUs for mobile SoCs today. To provide the efficient and convenient debug/trace environment to software developers, virtually all ARM processors including Cortex-A8, A9 and A15 embed the ARM CoreSight debug architecture [32]. The CoreSight architecture provides features for continuous collection of the processor execution traces using the hardware trace unit. Utilizing this unit, the hardware IPs proposed in the work can obtain the real-time traces of branch outcomes produced during code execution.

Although these approaches using the tracing hardware could achieve high performance in CRA detection, they are facing another challenging problem. In principle, in the debug environment using the hardware interface like CoreSight, it is assumed that the debugger has the same binary code running on the host. Thus, to reduce the quantity of traces delivered to the debugger, the interface generally does not provide the information which could be inferred or simply extracted from the binary code. However, unfortunately, these omitted pieces of information such as branch types or source addresses for branch instructions are indispensable for accurate CRA monitoring. To supplement the lacking information, in previous work, they store in the main memory region the auxiliary information, called the *meta-data*, that is necessary for CRA detection, and make the hardware IPs to read the data at runtime when the detection scheme needs to reference the data. In spite of the negligible performance overhead, they severely suffer from the substantial storage overhead due to the additional space for their meta-data. According to their experiments, the size of the required storage for meta-data can even be twice bigger than that of the original application. Another limitation of their hardware implementations is that they are only capable of detecting *return-oriented programming* (ROP) attacks, which corrupt return addresses stored in a stack to chain gadgets. Although ROP attacks are representative examples of CRAs, there is another breed of CRAs, called *jump-oriented programming* (JOP) attacks, whose objective is to alter the target addresses of indirect calls or jumps. To successfully defend the system against CRAs, therefore, the CRA monitoring hardware should be implemented with mechanisms that can detect not only ROP but also JOP attacks.

In this chapter, we present a hardware-based CRA solution that can simultaneously monitor both ROP and JOP attacks on the system. To underline the applicability of our solution to existing smart devices, this *unified* ROP/JOP monitor is implemented as IPs and integrated into an ARM-based SoC. As in previous work [1], the monitor is connected with the ARM CPU via the CoreSight interface and system bus to keep track of the host execution traces from outside in a timely fashion. In addition, for efficient monitoring, we have also made an effort to avoid substantial storage overhead due to meta-data in the previous work. For this, we analyze the program binary with the help of compiler analysis techniques and instrument the binary in a way that missing essential information for CRA monitoring can be efficiently delivered on the fly from the host CPU via the debug interface, thereby eliminating the need to store meta-data a priori for our monitor. However, a problem with this approach is that the two independent interfaces (i.e., the debug interface and the system bus) through which our external monitoring IPs receive host's runtime information are not perfectly synchronized; that is, when at some point in the code the CPU executes an instruction, proper pieces of the information for that execution will be generated and eventually transferred to our monitor through each interface, but not necessarily at the same time. Obviously, our monitor must correctly puzzle together these information pieces asynchronously arriving from two different sources to grasp the exact execution behaviors on the host for CRA detection. To resolve this issue, we added a special hardware logic to synchronize the incoming information from the two sources.

The rest of the chapter is organized as follows. Section 5.2.2 first describes some background information about the ARM architecture and how ROP/JOP attacks can be launched on ARM, and explains the threat model with our assumptions. After Section 5.3 presents the overall system architecture for the ROP/JOP detection, Section 5.4 explains in detail how ROP/JOP attacks are efficiently detected with help of the binary instrumentation in our approach. Also in this section, the detailed hardware architecture of our ROP/JOP monitoring modules will be explained. Then, Section 5.5 discusses the experimental setup and results. For the setup, we have used a ARM-based Zynq FPGA board [71] and prototyped our hardware modules to build a full SoC platform on the board. The results show that our prototype system offers a feasible security solution for protecting ARM-based SoCs against CRAs with high speed and low storage overhead. Finally in Section 5.6, we give some concluding remarks.

5.2 Background and Assumptions

To better understand our approach prototyped on the ARM-based platform, we will firstly provide a brief explanation on relevant aspects of the ARM processor architecture and introduce CRA samples targeting the ARM. After that, we will discuss the attack model we assume.

5.2.1 Background

5.2.1.1 ARM Processor Architecture

In the 32-bit ARM processor, 16 general-purpose registers (`r0-r15`) are provided. One noticeable feature of ARM architectures is that all ARM registers can be accessed directly by general instructions. Since the program counter PC is also aliased to a general-purpose register(`r15`), its value can be changed by various types of instructions including moves, arithmetics or loads/stores if it is used as a destination register. Therefore, the control flow of program can be modified by not only instructions of branch/jump type but also other types of instructions.

The ARM architecture provides two types of instruction sets, 32-bit

ARM and 16-bit/32-bit THUMB to support the systems with limited memory space. It is noticeable that many Linux libraries such as `libc` or `libwebcore` have accepted the use of THUMB instructions [39] to improve the code-density. Since attackers in general gather their gadgets from the library code which is widely available, most CRA gadgets targeting for ARM processors tend to contain many THUMB instructions [72]. To indicate which instruction-set is currently used, ARM has a status bit, called the *T-bit*, in current program status register (CPSR).

According to the ARM official document [73], function calls are implemented by `bl` (branch with link) or `blx` (branch with link and exchange) instructions. Both the instructions perform a branch with link operation that changes PC (or `r15`) while the return address is saved to the link register LR (aliased to `r14`). The `blx` instruction has another functionality. When bit 0 of the branch target address is set, `blx` sets the T-bit in CPSR to switch the instruction mode to THUMB. On the contrary, when the bit 0 is cleared, T-bit is also cleared to change the operating mode back to ARM. When attackers find their gadgets in the existing code base, this unique feature of `blx` can be exploited. Even when a code binary is written and compiled as 32-bit ARM by the programmer, the attackers still can forcefully read it as 16-bit THUMB code by setting the bit 0. Through this distorted code misuse, they are usually able to discover from the code base a plenty of gadgets consisting of THUMB instructions which were never meant to exist in the original code.

In the ARM architecture, any instruction that can take PC as its destination is able to be used as a return. The most common pattern of a return is to execute a `bx` (branch exchange) instruction with LR. The `"bx lr"` instruction replaces PC with the saved return address in LR (`r14`) while the T-bit can be also changed according to the address. Another way is to use the `ldm` (load multiple) or `pop` instructions that take PC as the destination operand. In this case, the return address which was pushed on the stack is restored to PC. In case of indirect jumps, the `bx` instruction is executed with the register operand storing the target address.

5.2.1.2 CRA Examples on ARM Processors

To launch an ROP attack, the adversary usually exploits software vulnerabilities (e.g., buffer overflow, integer overflow, or use-after-free) to overwrite a part of a data region in memory (e.g., heap, stack) which contains control data (e.g., return address, function pointer). Recall that in ARM, by writing a return address to PC, we can easily manipulate the target of a return to point to an arbitrary intended location within the existing code. This means that once vulnerability is found, the attacker may maneuver at her disposal the control flow transfers during code execution. These pointed snippets of code become the gadgets of a ROP attack which are linked together to construct a malicious program. If a sufficiently large code base is given for the attacker, the ROP attack model can be Turing-complete [74]. Capitalizing on this powerful expressiveness, ROP has been tried in many machine platforms such as SPARC [75], Atmel AVR [76], PowerPC [77] and ARM processors [78].

Even though the ROP attack model is capable of achieving Turing-complete computation, it requires tremendous efforts for adversaries to construct a complex program for an elaborated attack since they usually need to crawl painstakingly over a large code base in search of all necessary gadgets.

For this reason, the role of ROP is mainly confined to opening the door for the attackers to execute the injected code for real attack. To achieve this goal, they endeavor to build a ROP attack that invokes a system call which either opens a new shell or changes the attributes of memory pages to execute their injected code.

Figure 12 shows one such example of ROP attack implemented for the ARM processor. The purpose of this example code is to invoke a system call for a new shell with the command `"/bin/sh"` in Linux OS. To devise the ROP attack, we have gathered useful gadgets from `libc` library of ARM linux kernel 3.8. A part of the gadget chain is shown on the left of the figure. In ARM, as explained, there is no dedicated return instruction and thus in this example, `pop` instructions are used to conduct return operations. We have placed the return addresses inside the stack in order to chain the gadgets as shown in the figure. During execution, each gadget performs its primitive operation such as `add`, `str` or `mov`. Then, the intended return address is popped from the altered stack, and the control flow is transferred to the next gadget. At the end, in the last gadget, the `svc` instruction invokes a system call with the function arguments fabricated by the attacker. Then it finally opens a new shell to initiate the attacker's own code.

Since the ROP attack model was first introduced, a number of defense measures have been proposed to get rid of its threats by recognizing the behaviors of ROP code which are distinct from those of normal codes [72]. However, unfortunately, these defenses have soon been neutralized by a new class of CRAs that are based on JOP techniques [72, 79]. The key difference of JOP from ROP is that each gadget ends in an indirect jump operation, such

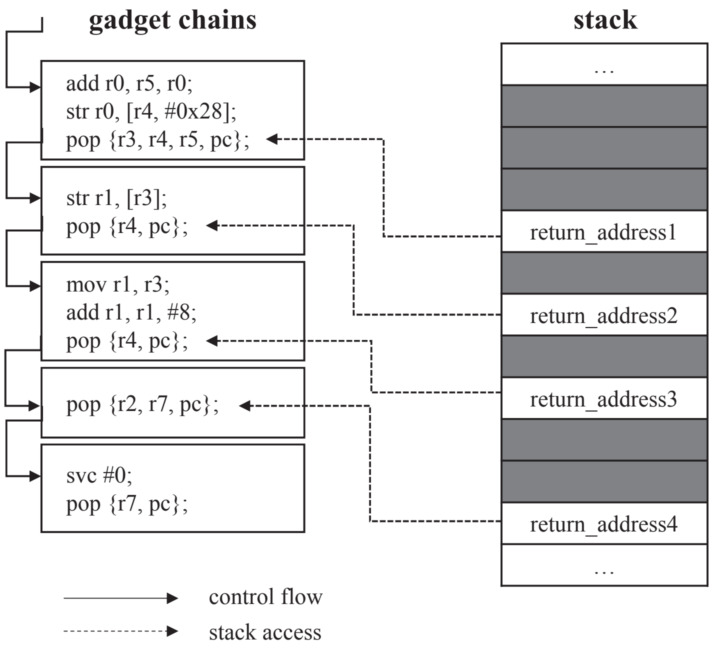


Figure 12: A ROP example on ARM processors

as `blx` and `bx` in ARM processors. Thus, the original defense mechanisms that attempt to capture the violations related to the uses of returns become useless. In ROP, a return instruction can freely access victim's stack to jump to the next code snippet in a gadget chain, but in JOP, an indirect jump has no such capability of accessing the stack. Therefore, the JOP attack model employs a special type of gadget, called the *dispatcher*, to create and manage a virtual stack from which the jump instruction in each gadget obtains the target address for the next gadget in a chain. For example, a code snippet `adds r6,#4;ldr r5,[r6,#124];blx r5` can be used as a dispatcher [72]. This gadget increments its virtual stack pointer register `r6` and then loads the next jump addresses from the stack. After jumping to the next gadget with the address in `r5`, the actual primitive operation is performed, and the control flow is transferred to the dispatcher again. In this manner, the model can gain its goal even without using return instructions.

5.2.2 Assumptions and Threat Model

We use the same assumptions on CRA taken by previous studies [70, 28]. We first assume that the target mobile device is under the protection of the W$\oplus$X policy and the OS is trusted. Considering that the modern OSes and processors usually cooperate to enforce the W$\oplus$X security protection rule [6], we believe this assumption is reasonable. Under this assumption, to circumvent the defense mechanism, the adversaries must gain sufficient privileges for the first time. We assume that, other than CRAs, there are no other attack vectors or security holes which can directly escalate adversary's privilege. As another assumption, adversaries might have full control over the stack or heap to exploit common memory vulnerabilities like buffer overflows and therefore can initiate a code-reuse attack. Also, the OS kernel and hardware are trusted until the underlying system is compromised through CRAs. We also assume that adversaries know all implementation details of the target application, thus being able to locate the exact address of available gadgets. This means that the adversary can bypass any code randomization techniques such as ASLR [5]. Lastly, the self-modifying code is not considered in our assumptions because it conflicts with the W$\oplus$X security protection.

5.3 Overall System Architecture

5.3.1 SoC Prototype Overview

Figure 13 depicts our overall SoC design. The monitoring modules for ROP/JOP detection were designed and implemented as a subsystem, called the *CRA monitor*, which is then integrated in a SoC platform with an ARM CPU. In our platform, the host CPU is an ARM Cortex-A9 processor, which has been installed in a large number of commercial devices these days. The host CPU and our monitor are connected via the standard AMBA3 AXI interconnect. To obtain the results of branch operations performed on the host, we utilize the built-in hardware modules of the ARM CoreSight debug architecture, which are the *program trace macrocell* (PTM) and the *trace port interface unit* (TPIU). Being tightly coupled with the host core, the two modules deliver the branch traces generated from the host to the CRA monitor.

It is noteworthy that, in terms of hardware design, the goal of our work is to build a practical and deployable hardware solution for CRAs on

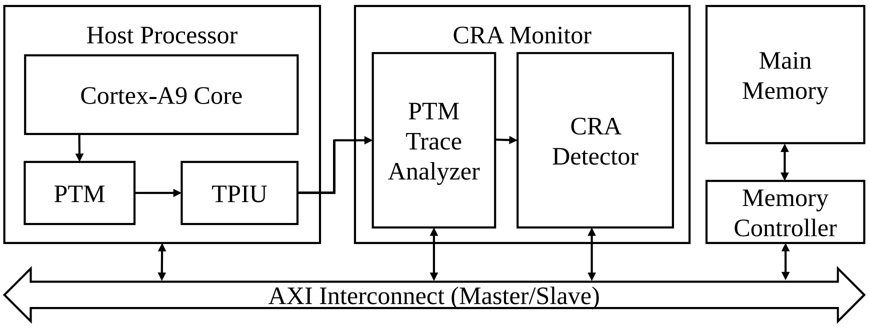


Figure 13: Overall architecture of our SoC design

ARM-based smart computing devices. To achieve the goal, we adhere to the design convention of the commercial SoC platforms, where off-the-shelf ARM processors and newly designed hardware modules are integrated and connected only through the existing communication channels, such as the system interconnect and the debug interface. As shown in the figure, our CRA monitor is divided into two modules: the *PTM Trace Analyzer* (PTA) and the *CRA detector*. To reduce the amount of transferred data, TPIU basically provides runtime traces in a highly compressed form. Thus, PTA analyzes and decompresses the incoming information from TPIU, and delivers to the CRA detector the refined branch traces which are necessary for the CRA detection. Upon receiving all the traces from PTA, the detector determines whether or not the traces exhibit any symptom of CRAs. In Section 5.4, more details of the hardware modules will be explained.

5.3.2 CRA Detection Process

As stated in Section 5.1, our CRA monitor detects both ROP and JOP attacks. To determine an attack from outside the host CPU, it must be provided with the necessary runtime information inside the CPU. In our work, to detect both types of attacks, we have realized in hardware the detection algorithms based on those proposed in [80] and [27], respectively. For ROP, we copy the return address of every call instruction in a special stack buffer called the *shadow stack* and check the target address of each return instruction with the value retrieved from the top of the shadow stack. Therefore, the necessary information to implement the shadow stack into our system are (1) the target address of return instructions and (2) the source address of call instructions to calculate the address to be returned later. Unlike ROP, JOP usually creates a code sequence by linking gadgets together with indirect jumps or calls. Hence, to launch JOP attacks, instead of altering return values stored in the stack, attackers try to corrupt code pointers such as function pointers, which will be used as the target addresses of indirect calls or jumps to point to their gadgets. The JOP detection algorithm is on the ground of a simple invariant ruling the normal behaviors of branches in a programming language. The invariant rule says that, in a normal program execution, the target address of a call instruction should point to the address of a function entry, and that of each indirect jump should always point to an address within the same function that the instruction belongs to. To check this legitimacy to detect JOP attacks, the CRA monitor has to obtain the information about (1) the target address of call instructions, (2) the target address of indirect jumps and (3) function boundaries which contain the entry and end addresses of functions. To summarize, the essential information to simultaneously check the existences of ROP/JOP attacks from outside the CPU is categorized into four classifications:

(1) Target address of indirect branches (i.e., indirect calls, indirect jumps and returns)

(2) Source address of call instructions

(3) Function boundaries

(4) Branch type to classify the branch instructions

Recall that, to reduce the quantity of generated traces, the ARM debug interface generally does not provide the information which can be directly derived from the binary code. In fact, only the target address of an indirect branch and the direction (taken/not taken) of a direct branch can be acquired from the traces coming through the debug interface. Gathering the target addresses of indirect branches are quite straightforward in our solution as the ARM debug interface is designed to provide such information. However, the other classes of information cannot be directly acquired from the debug interface, and therefore we have devised a special mechanism where we instrument the original binary to supply the lacking information. For this purpose, we built an in-house tool called the *binary instrumentor* that can statically instrument the target binary (phase 1). It basically analyzes and generates binary code in a way that all lacking pieces of the information for CRA detection will be explicitly delivered to the CRA monitor, either through the ARM debug interface or the system bus. When the program binary is downloaded by the OS kernel into the local storage such as a disk or a flash memory, the instrumentor generates the instrumented version of the binary and stores it into the storage. More detailed explanation will be given in Section 5.4. After the instrumented code is loaded, the CRA monitor performs its task of constantly watching the runtime traces gathered from both TPIU and the system bus and checking if there is any behavior possibly related to CRAs (phase 2).

5.4 IMPLEMENTATION DETAILS

5.4.1 Binary Instrumentation

As briefly discussed in Section 5.3, we propose a binary instrumentation technique that enables us to derive from the branch traces of the host system more information including not only the target addresses of indirect branches but also the branch types and the source addresses of call instructions. As the first step of the instrumentation, the binary instrumentor scans the entire code to find all function call instructions, which are executed by either a `bl` (branch with link) or a `blx` (branch with link and exchange) instruction in the ARM architecture. In order to deliver the information associated with the call instructions, we introduce a new code section called the *trampoline*. Each call instruction in the original code is moved to an associated location in the trampoline and the original instruction is replaced with an indirect jump which targets the associated place; specifically, each direct call (`bl` or `blx` with an immediate offset) moved to the trampoline is manipulated by the instrumentor so that it can target the same address as the original instruction pointing to. In addition, for each call in the trampoline, there is a unique stub which contains a direct jump to the next address of the original call. This stub is the target of the subsequent return instruction executed in the callee function. In Figure 14, we present an example to explain our instrumentation technique.

As shown in the Figure, when the address of the trampoline entry is A, every call instruction is aligned at addresses $A+8*n$, while the targets of return instructions are aligned at $A+8*n+4$ (n is an integer and $0 \leq n <$

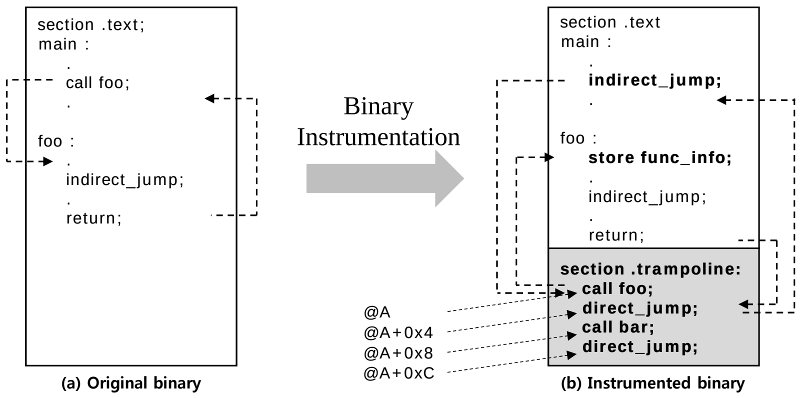


Figure 14: Original vs. instrumented binary (newly added parts are written in boldface)

total number of calls) . Using these aligned data, the types of the executed branch instructions can be classified by simply checking the target address coming from TPIU. Especially for a call instruction, an indirect jump to the trampoline can be followed either by a target address or by a direction (taken/not taken) information. When a target address follows the indirect jump, the branch type is considered to be an indirect call. Otherwise, it is decoded as a direct call. Note that all the calls are pointed to by indirect jumps as a result of the instrumentation. It means that the source address of each call can now be obtained from TPIU because the target address of the indirect jump pointing to $A+8*n$ becomes the source address of the call, allowing our monitor to calculate the legitimate destinations of return instructions which are necessary to maintain the shadow stack for ROP detection. Also, to detect JOP attacks, the function boundary information is indispensable to check if the target address of an indirect jump falls inside the function body where the current PC resides. Thus in our instrumentation scheme, each function is transformed in a way that it can start with an annotation code (`store func_info;` in Figure 14(b)) which writes the entry address and size of the function to the memory-mapped addresses of our hardware modules through the system bus. The binary instrumentor can identify the entry address and size of each function by referring to the symbol tables of executable formats such as executable and linkable format (ELF).

5.4.2 Hardware Architectures

Figure 15 shows the hardware structure of our CRA monitor including PTA. In our SoC prototype implementation, the output signals of TPIU are directly routed to the on-chip ports of our CRA monitor so that we can utilize the CoreSight modules. As the host CPU generally operates far faster than other hardware IPs such as our CRA monitor. Therefore, to transfer the PTM traces from the host to the monitor, we implement an asynchronous buffer, called the *branch trace FIFO*, which temporarily stores the traces coming from TPIU. When the traces are stored in the FIFO, another submodule in PTA called the *trace decoder* analyzes the saved traces to obtain the target addresses of indirect branch instructions and the direction (taken/not taken) of the direct branches. With this information, the decoder further extracts the branch types and source addresses of calls as mentioned in the previous subsection. Finally, for each branch instruction, its type and associate information (i.e., source addresses for calls and target addresses for indirect branches) are conveyed to the CRA detector for monitoring CRAs.

Figure 16 shows the unified hardware architecture of our CRA detector which keeps track of the host execution traces to simultaneously detect both

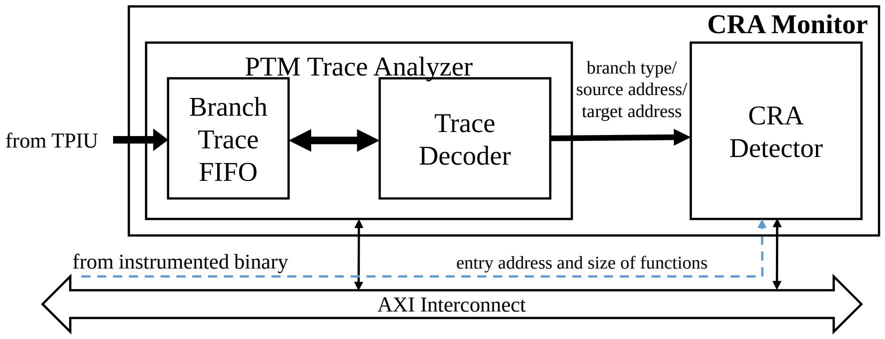


Figure 15: CRA monitor hardware architecture

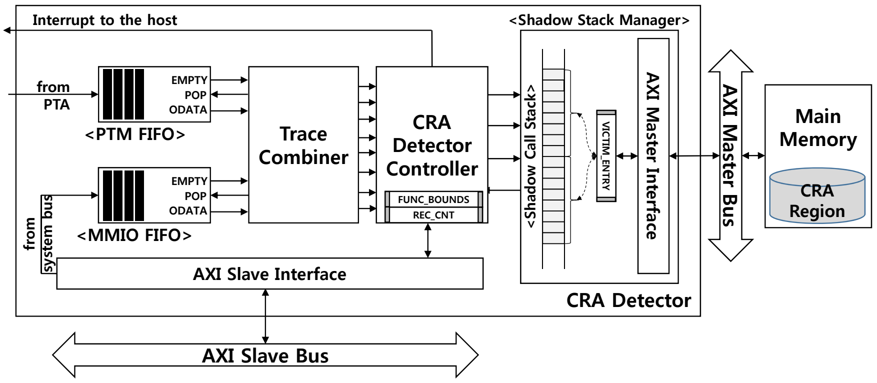


Figure 16: Hardware architecture of the CRA detector

ROP and JOP attacks. To find the existence of CRAs, our detector relies on the aforementioned branch information fed by PTA and the entry address and the size of functions coming through the system bus.

Recalling that the information from different sources (i.e., TPIU and the system bus) have no ordering restrictions, the CRA detector has to combine and rearrange the information from the two sources to keep track of the original program sequence of the application. In order to perform this task, the detector has two separate First-In-First-Out (FIFO) buffers, called the *PTM*

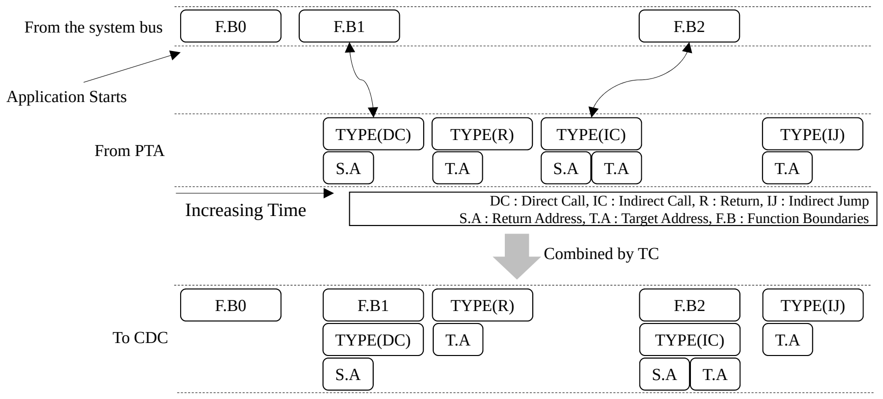


Figure 17: Information flow diagram processed by the Trace Combiner

FIFO and the *MMIO FIFO*, to temporarily store the information received respectively from PTA and the bus. The output signals of the FIFOs are given as input to the *trace combiner* (TC), which is in charge of combining the information from the two FIFOs and extracting the original program execution behaviors. We present the example of the information flow from the two sources and how they are combined by TC in Figure 17.

As exemplified in the figure, when the application begins, the program flow encounters the initially invoked function (i.e., main()) for the first time. This special event is notified to TC via the MMIO FIFO so that TC can start operation (presented in Figure 17 as F.B0). At runtime, when the program runs into a call instruction, the instrumented code at a function prologue is executed right after the call instruction, thus delivering the branch information (branch type and the associated information) and the function boundary information via the PTA and the system bus, respectively. When any of these events arrives and is stored in either the MMIO FIFO or the PTM

FIFO, TC reads it to take an appropriate action. If the function boundary information is written to the MMIO FIFO, TC waits for an event to come from the PTM FIFO. Once the PTM FIFO gets an entry, TC checks the branch type, and if it is either a direct or an indirect call, TC combines the pieces of information from the both FIFOs (i.e., the branch type, the function boundary information and the source address for a direct call as shown in Figure 17); otherwise, only the information from the PTM FIFO is selected. The information is then delivered to the *CRA detector controller* (CDC) whose mission is to make the final decision about the existence of CRAs.

After the application starts, CDC expects the information of the initially invoked function fed by TC before anything else. Upon receiving the information, CDC calculates the entry and end address (= entry address + size) of the callee function and stores them into a register called `FUNC_BOUNDS`. Later when the branch type coming from TC is a call, CDC also obtains the entry address and size of the callee function from TC. Especially for an indirect call, if its target address is not matched with the incoming function entry address, it means that the call jumped to an unknown address, which is a typical behavior exhibited by a JOP attack. If the call instruction is a direct one or verified to benign, CDC pushes the concatenated value of the return address (= source address + 0x4) and `FUNC_BOUNDS` onto the *shadow call stack*, whose job is to maintain a shadow copy of the call stack on the host. The reason why `FUNC_BOUNDS` is saved into the stack is that its value should be restored when the callee function returns later. At the same time, CDC overwrites `FUNC_BOUNDS` with the newly calculated entry and end addresses of the callee function. When a function returns, CDC pops the top entry of the stack and compares the saved return address against the target address coming from TC. If there is a mismatch, it means that the return address in the host stack is maliciously manipulated by ROP attacks, and consequently CDC issues an interrupt. Otherwise, CDC overwrites `FUNC_BOUNDS` again with the saved function boundaries. When an indirect jump is made in the host, CDC will check whether or not its target address falls between the entry and end addresses of the currently running function by referring to `FUNC_BOUNDS`. If the address points to outside the function boundaries, CDC deems that this is the act of a JOP attack, and spontaneously notifies the host of this attack by setting the `interrupt` signal on.

Note that the shadow call stack has a finite number of entries, 16 in this work. Therefore, it would be overflown if the target application has more than 16 times nested function calls. To cope with this limitation, we implemented a special stack management module called the *shadow stack manager* (SSM). When the shadow call stack fills up with deeply nested calls, SSM copies the oldest 8 entries to the pre-defined region, called the *CRA region*, in the main memory through the *AXI Master Interface* in SSM. Also, we implemented a register called `VICTIM_ENTRY` which plays a role as a victim cache storage to temporarily store the most recently evicted 8 entries. Moreover, there is an exceptional case that the host program calls the same function recursively. For handling this case, CDC has a counter register, which we refer to as `REC_CNT`, to store the number of recursive calls. When the same function is called in a row, CDC increase the counter value by one without pushing any value onto the stack. When the function returns and `REC_CNT` has a non-zero value, CDC decreases the counter value by one instead of reading the top stack value.

5.5 EXPERIMENTAL RESULTS

To evaluate our approach, we implemented a full SoC prototype on the Xilinx Zynq-7000 XC7Z020 evaluation board, which is equipped with a dual-core ARM Cortex-A9 processor, AMBA3 AXI interconnect, 1GB DDR3 SDRAM, an FPGA chip and other peripherals. We used Linaro Ubuntu Linux version 3.8.0 as our host kernel. Also, we enabled the CoreSight modules (i.e., PTM and TPIU) in the host processor and controlled them with the device driver which is extended according to our purpose. Our CRA monitor and the host CPU commonly operate at 60 MHz. Based on the above design parameters for the prototype, we synthesized the CRA monitor onto the FPGA chip and measured the required logic count in terms of lookup tables for logic (LUTs) and memory elements. The synthesis result shows that our CRA monitor occupies 10.12% (5,387/53,200) of total LUTs and 0.13% (24/17,400) of total memory elements.

To measure the detection capability of our monitor, we implemented five CRA instances based on the Shell-storm shellcode [81] as shown in Table 9. Especially, A2 and A5 contain long-gadgets to bypass the signature-based CRA solutions proposed in [28, 70], which use the small number of instructions in a gadget as the distinctive feature of CRAs. libraries (i.e., libwebcore in Android 4.2.2, libc-2.13 in Xilinx-linux and libc-2.15 in Ubuntu) as our code base.

With the implemented attacks, we tested the detection capability of our

Attack No.	Type	Goal	Advanced Skill	Detection
A1	ROP	Open a shell	-	√
A2	ROP	Open a shell	Long-gadget	√
A3	ROP	Invoke a mprotect system call	-	√
A4	JOP	Open a shell	-	√
A5	JOP	Open a shell	Long-gadget	√

table 9: The description of implemented CRAs and detection results of the attacks

monitor. As expected, all the ROP samples (A1-A3) are detected by our CRA monitor. Since they violate the general convention of the function invocation, their malicious behaviors are detected by our CRA monitor even when the attacks contain long-gadgets which is an advanced skill for circumventing the state-of-the-art CRA detection schemes. The JOP samples (A4-A5) are crafted by using `blx` (indirect call) or `bx` (indirect jump) instructions of ARM ISA to link their gadgets. In these attacks, every `blx` instruction used to link gadgets does not target an entry of a function. Similarly, all the target addresses of `bx` instructions are always beyond the current function bounds. Consequently, all their illegal behaviors are detected by our CRA monitor. Based on this result, we assert that our CRA monitor can protect the target system from any type of CRAs.

To measure the performance overhead of our CRA monitor, we chose eight applications from the SPEC CPU2006 benchmark suite [57]. We compared the running time for the applications using two configurations. The first one is *Base* which acts as the control group where the execution of the original code runs on the host processor with the CRA monitor disabled, thus being exposed to CRA attacks. The other is *wCRA* that refers to the same

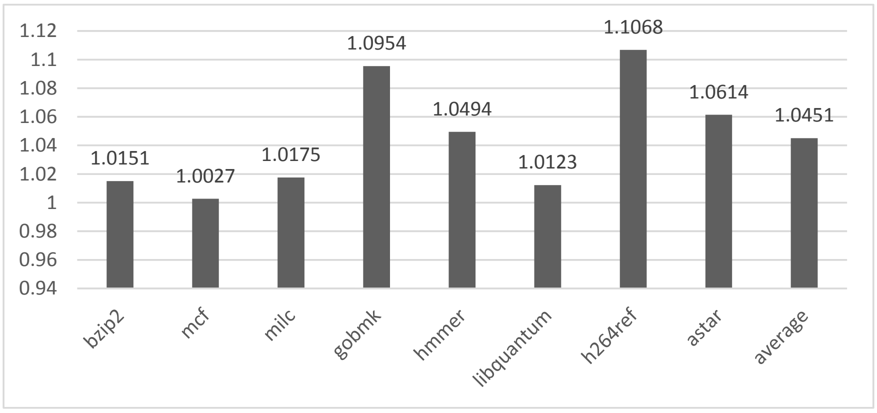


Figure 18: Benchmark execution time when the CRA monitor is enabled

code execution with the CRA monitor enabled. We show the performance numbers of *wCRA* in Figure 18 where the execution time of each application with *wCRA* is normalized to that of *Base*. The empirical results show the running time overhead of 4.51%/10.68% (average/max) over *Base*.

Also, we compared the storage overhead due to our instrumentation with the overhead incurred by the meta-data proposed in [1]. Even though the meta-data has been introduced to accelerate the overall detection process, it induces substantial storage overhead proportional to the code size of the target application. Although our approach also requires the binary code running on the host CPU to be instrumented with additional instructions, we argue that the amount of additional code is rather small compared to the previous approaches. To support this argument, we measured the amount of memory required for the CRA detection suggested in [1] and ours, as presented in Table 10. As seen in the figure, our approach needs slightly more memory than the original, uninstrumented code, but requires far less

Benchmark	Original size (a)	Ours		[8]	
		increased code (b)	(b)/(a)	meta-data (c)	(c)/(a)
bzip2	503,664	88,020	0.1748	797,144	1.5827
mcf	464,775	82,836	0.1782	725,992	1.5620
milc	588,408	114,564	0.1947	1,169,487	1.9875
gobmk	3,973,190	286,032	0.0720	1,856,400	0.4672
hmmer	764,042	156,472	0.2048	1,147,512	1.5019
libquantum	561,254	96,804	0.1725	863,652	1.5388
h264ref	1,000,235	143,916	0.1439	1,474,460	1.4741
astar	579,187	107,604	0.1858	885,456	1.5288
average			**0.1658**		**1.4554**

table 10: Comparison of binary sizes between ours and [1]

memory (on average 16.58%) than that of the technique proposed in [1] (on average 145.54%).

The above results clearly show the advantage of our approach over the previous work [1] in terms of memory usage. The removal of the meta-data also gives us another advantage that our hardware IPs no longer need to read a large quantity of data from the main memory at runtime. Although the experiment in [1] reported that their performance overhead is about 3% due mainly to memory contention between the host and their monitor, which is slightly better than ours, we have discovered that their approach relying on massive memory accesses for meta-data inherently entails a serious flaw. In their work, the latency to the main memory such as DDR has to be paid for processing each branch trace coming from the debug interface. Since it requires the reference to the meta-data, the processing capability of the monitoring hardware is severely limited. This trend gets more obvious when the user wants to increase the CPU frequency for the higher host performance or decreases the DDR frequency for the less power consumption.

ARM_CLK:IP_CLK	1:1	2:1	3:1	4:1	5:1
Ours	o	o	o	o	o
[1]	o	x	x	x	x

table 11: Frequency gap tolerance of ours and [1]
(IP_CLK is for both the monitor and the DDR memory)

In these conditions, branch traces are more likely to be dropped without being analyzed. To put forward evidence to support the hypothesis, we measured the operable frequency gap between the host CPU and our CRA monitor. For this experiment, we implemented the ROP monitor in [1] and checks how slow their monitor can operate while correctly performing the CRA detection. Then, we compared the result with that of ours in Table 11. Both of them are configured to have the same depth of the input buffers (32 in this experiment) to temporarily store the incoming traces from TPIU. As we expected, ours tolerates up to the 5:1 frequency gap without overflowing the buffer. On the other hand, the work in [1] cannot stand even the 2:1 frequency gap. This result indicates that their solution does not function correctly for more realistic SoC architecture models where the host CPU is much faster than external devices like our monitor. In this sense, we believe that our approach is more acceptable in real-world systems such as APs of modern smartphones [82] whose frequency gaps between the host CPU and other auxiliary IPs are typically configured up to 5:1.

5.6 Conclusion

We have discussed how our hardware solution has been integrated into an ARM-based SoC to defend the system against ROP and JOP attacks at the same time. The solution incurs very low performance overhead for runtime detection of CRAs by implementing the unified hardware IPs to efficiently detect both types of attacks. Our solution does not require any modification in the host ARM processor internal. Therefore, our hardware modules can be easily integrated with a commodity ARM processor core, observing the conventional SoC design rules so that our solution can be easily implanted to commercial mobile SoCs. Moreover, our key contribution is that ours reduces the storage overhead dramatically compared to the previous work. To achieve this, we propose an instrumentation technique which enables us to make the most use of the existing debug interface. The experiments revealed that our current implementation successfully detects synthetic ROP/JOP attacks, and that the storage overhead incurred by our solution is acceptably small when being compared to the previous work.

Chapter 6

Conclusion

This thesis presents a new security monitoring solution in which an external hardware monitor is deployed in the system and connected to the host processor via a standard debug interface, CDI, so that the monitor can be fed with bountiful information generated from the host. Our solution inherits the advantage of recently proposed hardware-based monitoring solutions in that the computation-intensive tasks are offloaded to the specialized external hardware engine, thus reducing the burden of the host processor introduced by the monitoring tasks. In addition, our solution establishes a separate communication channel between the host and the external monitor via CDI, which provides, at real-time, the monitor with information relavant to the control-flow and/or data-flow of the target program running on the host. This functionality provides a key to design various security monitors that can attain both high level of security and low performance overhead, without necessity of changing the host system internals.

To show the validity of our solution, in this thesis, we realized three security monitoring systems each of which addresses different types of attacks. The first system concerns the cache resident attack that intends to break the integrity of an OS kernel. Under the vigilance of our proposed snoop-based monitor and thanks to the data-flow information transferred via CDI, no cache resident attack can succeed in compromising the kernel. The second system uses the control-flow information coming from CDI to detect any suspicious activities that violate a set of rules which code-reuse attacks (JOP/ROP) may break. With the information, our system can successfully detect majority JOP and ROP attacks, except for the newly developed *call-oriented programing* style attacks, which no known defense mechanism has succeeded in detecting. The last system uses both control-flow and data-flow information to monitor the trace of secret data. At runtime, every data derived from the secret data is tainted and tracked throughout the operation, and if any tainted data is involed in potentially illegal activities, our monitoring system raises an alarm.

The experiment results in each section evince that, implemented and verified on a FPGA prototyping board, the solutions can successfully catch various attacks deployed on the system. Moreover, the performance estimation with a group of benchmark applications shows that our systems do not suffer from severe performance loss. The experiment results also reveal that the area overhead of the hardware is acceptably small when compared to the normal sizes of today's mobile processors. We believe that our solution can be applied to establish more wide range of security monitoring schemes, which need to observe the processor internal states to detect malicious behaviors of attacks.

Reference

[1] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, “Towards a practical solution to detect code reuse attacks on arm mobile devices,” in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, p. 3, ACM, 2015.

[2] ISO, “Trusted Platform Module,” 2009.

[3] LINUX, “Address Space Layout Randomization,” 2009.

[4] MS, “Address Space Layout Randomization,” 2007.

[5] PaX Team, “Address Space Layout Randomization,” 2003.

[6] S. Andersen and V. Abella, “Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies,” 2004.

[7] Y. Bulygin and D. Samyde, “Chipset based approach to detect virtualization malware,” *BlackHat Briefings USA*, 2008.

[8] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, “Ensuring operating system kernel integrity with osck,” in *ACM SIGPLAN Notices*, vol. 46, pp. 279–290, ACM, 2011.

[9] Z. Wang and X. Jiang, “Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity,” in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 380–395, IEEE, 2010.

[10] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, “Vigilare: toward snoop-based kernel integrity monitor,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 28–37, ACM, 2012.

[11] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "Cpu transparent protection of os kernel and hypervisor integrity with programmable dram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pp. 392–403, ACM, 2013.

[12] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "Ki-mon: a hardware-assisted event-triggered monitoring platform for mutable kernel object," in *USENIX conference on Security*, USENIX Association, 2013.

[13] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot-a coprocessor-based kernel runtime integrity monitor.," in *USENIX Security Symposium*, pp. 179–194, 2004.

[14] N. L. Petroni Jr and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 103–115, ACM, 2007.

[15] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 340–353, ACM, 2005.

[16] W. Cheng *et al.*, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *ISCC '06*, 2006.

[17] F. Qin *et al.*, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *2006. MICRO-39.*, 2006.

[18] G. E. Suh *et al.*, "Secure program execution via dynamic information flow tracking," ASPLOS XI, ACM, 2004.

[19] M. Dalton *et al.*, "Raksha: a flexible information flow architecture for software security," ISCA '07, ACM, 2007.

[20] S. Chen *et al.*, "Flexible hardware acceleration for instruction-grain program monitoring," ISCA '08, 2008.

[21] V. Nagarajan *et al.*, “Dynamic information flow tracking on multicores,” 2008.

[22] J. Newsome *et al.*, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” in *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[23] P. Chen *et al.*, “DROP: Detecting return-oriented programming malicious code,” in *Information Systems Security*, pp. 163–177, Springer, 2009.

[24] A. Baliga, V. Ganapathy, and L. Iftode, “Automatic inference and enforcement of kernel data structure invariants,” in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pp. 77–86, IEEE, 2008.

[25] N. L. Petroni Jr, T. Fraser, A. Walters, and W. A. Arbaugh, “An architecture for specification-based detection of semantic integrity violations in kernel dynamic data,” in *USENIX Security Symposium*, 2006.

[26] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Memtracker: Efficient and programmable support for memory access monitoring and debugging,” in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 273–284, IEEE, 2007.

[27] M. Kayaalp *et al.*, “Branch regulation: Low-overhead protection from code reuse attacks,” in *Computer Architecture (ISCA), International Symposium on*, pp. 94–105, June 2012.

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

[29] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, “HAFIX: Hardware-assisted flow integrity extension,” in *Proceedings of the The 52nd Annual Design Automation Conference on Design Automation Conference*, pp. 1–6, June 2015.

[30] Z. Guo, R. Bhakta, and I. G. Harris, “Control-flow checking for intrusion detection via a real-time debug interface,” in *Smart Computing Workshops (SMARTCOMP Workshops), 2014 International Conference on*, pp. 87–92, IEEE, 2014.

[31] H. Kannan *et al.*, “Decoupling dynamic information flow tracking with a dedicated coprocessor,” in *DSN ’09.*, 2009.

[32] ARM co., LTD, “ARM CoreSight Architecture Specification v2.0,” 2013.

[33] Intel, “Intel 64 and IA-32 Architectures Software Developer Manuals.”

[34] Xilinx, *MicroBlaze Processor Reference Guide*, Apr 2012.

[35] W. Orme, “Debug and trace for multicore socs,” Sep 2008.

[36] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” CCS ’07, ACM, 2007.

[37] Y. Zhu, J. Jung, S. Dawn, T. Kohno, and D. Wetherall, “Privacy scope: A precise information flow tracking system for finding application leaks,” in *Tech. Rep. EECS-2009-145, Department of Computer Science, UC Berkeley*, 2009.

[38] N. Nethercote *et al.*, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” PLDI ’07, ACM, 2007.

[39] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, “Mocfi: A framework to mitigate control-flow attacks on smartphones.,” in *NDSS*, 2012.

[40] G. Venkataramani *et al.*, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *HPCA*, 2008.

[41] I. Heo, M. Kim, Y. Lee, C. Choi, J. Lee, B. B. Kang, and Y. Paek, "Implementing an application-specific instruction-set processor for system-level dynamic program analysis engines," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 4, p. 53, 2015.

[42] D. Hollingworth and T. Redmond, "Enhancing operating system resistance to information warfare," in *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, vol. 2, pp. 1037–1041, IEEE, 2000.

[43] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pp. 239–242, ACM, 2002.

[44] M. Rebaudengo and M. S. Reorda, "Evaluating the fault tolerance capabilities of embedded systems via bdm," in *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pp. 452–457, IEEE, 1999.

[45] J. Peng, J. Ma, B. Hong, and C. Yuan, "Validation of fault tolerance mechanisms of an onboard system," in *Systems and Control in Aerospace and Astronautics, 2006. ISSCAA 2006. 1st International Symposium on*, pp. 5–pp, IEEE, 2006.

[46] A. V. Fidalgo, G. R. Alves, and J. M. Ferreira, "Real time fault injection using a modified debugging infrastructure," in *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, pp. 6–pp, IEEE, 2006.

[47] M. Portela-García, C. López-Ongil, M. G. Valderas, and L. Entrena, "Fault injection in modern microprocessors using on-chip debugging infrastructures," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 2, pp. 308–314, 2011.

[48] M. Grosso, M. S. Reorda, M. Portela-García, M. García-Valderas, C. López-Ongil, and L. Entrena, “An on-line fault detection technique based on embedded debug features,” in *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*, pp. 167–172, IEEE, 2010.

[49] M. Gallardo-Campos, M. Portela-Garcia, C. Lopez-Ongil, L. Entrena, M. Grosso, and M. S. Reorda, “Enhanced observability in microprocessor-based systems for permanent and transient fault resilience,” in *Conference on Design of Circuits and Integrated Systems (DCIS)*, pp. 240–246, 2010.

[50] M. Portela-García, M. Grosso, M. Gallardo-Campos, M. Sonza Reorda, L. Entrena, M. García-Valderas, and C. López-Ongil, “On the use of embedded debug features for permanent and transient fault resilience in microprocessors,” *Microprocessors and Microsystems*, vol. 36, no. 5, pp. 334–343, 2012.

[51] “The blue pill, doi=http://theinvisiblethings.blogspot.com/2008/07/0wing-xen-invegas.html,”

[52] “Vmware: Vulnerability statistics,”

[53] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, “Dksm: Subverting virtual machine introspection for fun and profit,” in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pp. 82–91, IEEE, 2010.

[54] Aeroflex Gaisle, *GRLIB IP Core User’s Manual*, January 2012.

[55] ARM, *Embedded Trace Macrocell Architecture Specification*, 2011.

[56] SPARC International, Inc., *The SPARC Architecture Manual*, 1992.

[57] J. L. Henning, “Spec cpu2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[58] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[59] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, “Secure in-vm monitoring using hardware virtualization,” in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 477–487, ACM, 2009.

[60] A. ARM, “Architecture reference manual (armv7-a and armv7-r edition),” *ARM DDI C*, vol. 406, 2008.

[61] I. Intel, “http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.html,” *Intel64*, vol. 406, 2014.

[62] “Leon3 processor user’s manual, gaisler research,” 2004.

[63] ARM, *AMBA Specification*, 1999.

[64] D. C. U. Guide, “Version c-2009.06,” *Synopsys.(a)(b)(c)*, 2009.

[65] S.-W. Olle *et al.*, “Evaluation of the energy efficiency of arm based processors for cloud infrastructure,” *Turku Centre for Computer Science*, 2010.

[66] M. Graphics, “Modelsim,” 2007.

[67] K. Lab, “Dorifel Malware Encrypts Files, Steals Financial Data, May Be Related to Zeus or Citadel,” 2012.

[68] M. Guthaus *et al.*, “Mibench: A free, commercially representative embedded benchmark suite,” in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, 2001.

[69] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th*

ACM conference on Computer and communications security, pp. 552–561, ACM, 2007.

[70] V. Pappas *et al.*, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proceedings of the 22Nd USENIX Conference on Security*, pp. 447–462, USENIX Association, August 2013.

[71] Features, ZC702 Board, "ZC702 evaluation board features," *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 Extensible Processing Platform*, 2012.

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

[73] ARM co., LTD, "Procedure call standard for the arm architecture," 2012.

[74] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Recent Advances in Intrusion Detection*, pp. 121–141, Springer, 2011.

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

[76] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 15–26, ACM, 2008.

[77] F. Lindner, "Developments in cisco ios forensics. confidence 2.0," 2009.

[78] T. Kornau, "Return oriented programming for the arm architecture," *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[79] T. Bletsch *et al.*, “Jump-oriented programming: A new class of code-reuse attack,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 30–40, ACM, March 2011.

[80] H. Özdoganoglu, T. Vijaykumar, C. E. Brodley, B. Kuperman, A. Jalote, *et al.*, “Smashguard: A hardware solution to prevent security attacks on the function return address,” *Computers, IEEE Transactions on*, vol. 55, no. 10, pp. 1271–1285, 2006.

[81] The shell storm linux shellcode repository, 2014.

[82] Samsung Electronics co., LTD, “Exynos,” 2015.

초 록

하드웨어 기반 보안 모니터링을 위한 호스트 시스템의 정보 추출

이 진 용
전기컴퓨터공학부
서울대학교 대학원

기술이 발전함에 따라 우리의 삶에서 전자기기들이 차지하는 비중이 점점 증가하고 있다. 이러한 현상은 IoT 환경이 도래함에 따라 더욱 가속화될 것으로 예상되며 이에 따라 이러한 기기들이 다루게 될 민감한 정보의 양도 크게 증가할 것으로 예상되고 있다. 이러한 정보들을 보호하기 위해 다양한 관점에서 디바이스의 보안성을 높이려는 연구들이 진행되어 왔지만 이들 중 극소수의 연구들만이 실제 기기들에 적용이 되었고 대부분의 연구들은 가능성만을 보이는데 그쳐왔다. 이는 제안된 보안 메커니즘이 실제로 우리가 사용하는 기기들에 빠르게 적용되기 위해서는 전체 시스템에 미치는 성능 저하가 충분히 작아야 할 뿐만 아니라 전체 시스템의 디자인 시간 및 비용의 절감을 위해 감시 대상이 되는 호스트 프로세서의 구조를 크게 변경하지 않는 방법이어야 하기 때문이다. 최근 제안되는 하드웨어 기반의 보안 모니터링 시스템은 높은 보안성을 자랑하는 동시에 모니터링에 특화된 하드웨어의 사용을 통해 전체 시스템의 성능 저하도 최소화 할 수 있다는 점에서 각광을 받고 있다. 하지만 최근 발표된 연구의

결과에 따르면 모니터링을 위한 하드웨어가 호스트 프로세서의 내부에 밀접하게 결합되어 있지 않은 경우 모니터 하드웨어가 호스트 프로세서에서 수행되고 있는 코드의 실행 정보 및 호스트의 상태를 제대로 알 수가 없기 때문에 모니터링할 수 있는 이벤트가 크게 줄어들어 모니터링 하드웨어의 효용성이 크게 떨어질 수 밖에 없게 된다. 이를 피하기 위해 호스트 프로세서와 모니터링 하드웨어가 메모리 등의 자원을 공유하는 형태로 정보를 주고 받게 되면 빈번한 정보의 교환 때문에 상당한 성능 저하가 발생하게 된다. 이러한 문제를 해결하기 위해 우리는 이 논문에서 새로운 보안 모니터링 솔루션을 제안한다. 이 솔루션은 이전에 제안된 보안 모니터링 기법들과 마찬가지로 외부 하드웨어 기반의 모니터의 이용을 제안한다. 하지만 이에 추가적으로 이 솔루션은 호스트와 모니터링 하드웨어 간의 정보 교환을 위해 공유 메모리와 같은 시스템 자원을 이용하기보다는 최신 프로세서에 이미 탑재되어 있는 디버그 인터페이스를 이용하는 것을 제안한다. 디버그 인터페이스는 호스트 프로세서의 코드 수행 정보 및 메모리 접근에 대한 정보를 디버깅 목적으로 외부로 전송할 수 있는 기능을 가지고 있다. 이 디버그 인터페이스에 연결되게 되면 외부 모니터 하드웨어는 본래 디버깅 용으로 만들어진 다양한 정보를 받아 보안 모니터링에 사용할 수 있는 가능성을 갖게 된다. 본 논문의 3-5장에서는 제안된 모니터링 솔루션의 유효성을 체크하고 디버그 인터페이스로부터 전달받은 여러 정보들을 이용해 보안 모니터를 구현하게 되는 경우 어떠한 점들이 고려되어야 하는지 알아보기 위해 세 가지의 잘 알려진 보안 문제를 풀 수 있는 외부 하드웨어 기반 모니터링 시스템을 구축해 그 결과를 보여준다. 이러한 보안 모니터링의 실제 구현을 통해 우리는 우리가 제안한 솔루션, 즉 디버그 인터페이스를 이용해 보안 모니터링 시스템을 구현하는 경우 높은 성능 및 보안성 뿐만 아니라 호스트 프로세서의 내부를 수정하지 않

는다는 장점을 갖는다는 것을 보여줄 수 있었다.

키워드 : 정보 보안, 하드웨어 기반 보안 모니터링, 디버그 인터페이스

학 번 : 2011-30250