Ph.D. DISSERTATION

Hardware Techniques against Memory Corruption Attacks

메모리 변조 공격 대응을 위한 하드웨어 기술

BY

Hyungon Moon

February 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Abstract

Many programs are written in unsafe languages like C or C++ mainly due to their advantages in performance, and most of them are too complex to be implemented without implementation errors. For these two reasons, such programs inevitably have vulnerabilities with which attackers can access their memory arbitrary. Unfortunately, it is said to be impossible to eliminate the vulnerabilities. Whereas programs can be verified not to have some vulnerabilities, only small programs can be analyzed statically and not all vulnerabilities can be found and fixed. To address the problem of the vulnerable programs, researchers have proposed a number of mechanisms to mitigate the attacks exploiting the vulnerabilities.

This thesis presents novel hardware-assisted mechanisms against those attacks exploiting the vulnerabilities, which are called the memory corruption attacks.

The first half discusses the ones against the attacks on OS kernels. In most computer systems, OS kernels have the full control. Every program running on a system has to call the kernel to access or acquire the resources of the system such as the network, file system, or even the memory. This nature makes the OS kernels be an attractive target for attackers. Taking control of it, they can affect every single program running on the system.

A difficulty in devising mechanisms to mitigate the attacks on OS kernels comes from the fact that they control the system. Any mechanism that relies on the OS kernels can be nullified by the attackers with the control of the kernels. This lead to the research on the mechanisms that do not rely on the OS kernels themselves. This thesis presents the state of the art of the mechanisms using physically isolated hardware components to avoid relying on the OS kernels. We designed and implemented a novel means for such mechanisms to collect the kernel events efficiently and effectively, and utilized them to mitigate the common types of attacks.

The second half presents hardware-assisted mechanisms for memory corruption attacks in general. Though many mechanisms have been proposed to mitigate memory corruption attacks, most of them are not practical. Some of them have limited backward compatibility which requires the existing programs to be fixed to adopt them, and most of them are not efficient enough to be widely deployed.

This thesis aims to design practical mechanisms to mitigate memory corruption attacks, and presents two of such mechanisms. The first one enables the programs to isolate the data-flow of sensitive data from the others. Such isolation makes it more difficult for the attackers to corrupt the sensitive data because only the vulnerabilities in the code blocks accessing them can be exploited to corrupt them. The second one prevents the attackers from building up the attacks reliably by randomizing data space. Once a program adopts the mechanism, only the memory accesses complying with the results of the static analysis can be completed correctly. As the attacks usually cause the victim programs to violate the results, the attacker-induced memory accesses will cause unpredictable values to be stored or loaded.

In summary, this thesis presents four mechanisms to mitigate the memory corruption attacks either on OS kernels or user-level programs.

keywords: Memory Corruption Attacks, Operating System Kernel,
Code-Injection Attack, Core-Reuse Attack, Data-Flow Integrity,
Snoop-Based Monitoring
student number: 2010-20795

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many programs have known or unknown vulnerabilities which attackers can exploit to corrupt their memory contents. The attackers corrupt the programs' memory in a variety of ways and purposes. Some of them modify the programs' data only. Modifying a particular data, for instance, the attackers can mislead a server to allow a malicious user to log in. Other attackers mislead the programs to execute their code at certain moments such as when they handle a particular user input, when they wake up, or whenever they communicate with a specific server. As a result, the victim programs will handle such events in attacker-directed ways.

Unfortunately, such programs are usually implemented in unsafe languages like C or C++. Though small ones could be verified statically that they are free from some vulnerabilities, most of them are not small enough to be verified. For this reason, new vulnerabilities of such programs are reported every year. Although they are patched soon after being reported, attackers would be able to compromise such programs with unpublished ones that they found, even when a system runs fully patched programs only.

For decades, a number of mechanisms have been proposed to mitigate such *memory corruption attacks* [1]. Some of them aimed at detecting or preventing the attacks at early stages, when they violate certain rules that a program is expected to follow.

Common rules that they used are the *memory safety* and *Data-Flow Integrity* (DFI). The ones enforcing the former checks if, for example, the pointers to arrays remain bounded or the ones to heap object used after being freed. The latter makes the memory instructions to access the objects which they are supposed to access so that the program's data-flow complies with the data-flow graph built statically. Others focus on the attacks that alter the execution flow of the victim programs. Earlier ones were designed to detect or prevent the *code-injection attacks* where the attackers put their code into the program's memory and mislead them to execute it. As such mechanisms are effectively preventing the injection attacks in most cases, recently proposed ones aim to mitigate the *code-reuse attacks* where the attackers build their code without injecting it into the program's memory.

This thesis presents novel architectural supports to mitigate these memory corruption attacks.

The first half of the thesis shows the mechanisms with the architectural supports that mitigate the attacks on OS kernels. Like any large programs written in unsafe languages, OS kernels could have a vulnerability and cannot be free from it. For instance, new vulnerabilities for the Linux kernel, which is used as the kernel of a dominant OS in the smartphone market [2], are reported every year [3, 4]. The mechanisms presented in this thesis make use of novel means of monitoring the kernel events to mitigate the attacks on it.

The second half is about the mechanisms against memory corruption attacks in general, which could mitigate the attacks on either OS kernels or user-level applications. While a number of mechanisms against the memory corruption have been proposed, they still are not practical due to their efficiency, effectiveness, and backward compatibility. To address the problems, the mechanisms presented in this thesis enforce DFI with novel architectural supports. To the best of our knowledge, they were the first attempts to design the specialized hardware modules for the DFI enforcement.

1.1 Hardware-based Monitors for OS Kernels

Many electronic devices have a sort of Operating Systems (OSes) which are responsible for managing their resources. At the center of the OSes are the OS kernels which access the system resources directly. With the privilege higher than that of any other programs on the devices, the OS kernels are the only program that can access all resources directly, and the other programs should ask the kernels for those resources. For this reason, once the OS kernel is under the control of the attackers, they can affect the behavior of any program running on it. For instance, they can manipulate the contents of file only when it is being read, drop every network packet to a certain destination, or prevent a particular program from using more than one processor core to run.

Due to the importance of OS kernels as the system's *Trusted Computing Base* (TCB), a number of remedies have been proposed against these attacks in a variety of aspects. Some of them implemented a sort of *monitors* using a software or hardware entity that runs without relying on OS kernels. In this way, the monitors could run as designed even when the victim kernel has already been compromised arbitrarily. Other mechanisms are designed to *harden* the kernels so that they become more resilient against a set of attacks. Though these could be bypassed or nullified by the attacks that they are designed to mitigate, they were effective against the targeted attacks. Among these two approaches, the mechanisms presented in the first half of this thesis follow the former, to detect the attacks by monitoring the kernel events.

Many kernel-independent monitors are designed to rely on *hypervisors*, which is also known as *Virtual Machine Monitors* (VMMs) [5]. Being designed to manage and provide resources for multiple virtual machines running their own OS kernels, hypervisors do not rely on them in general, and could have been extended to watch and examine their behavior. However, the hypervisors again are software components which could also have vulnerabilities, and more vulnerabilities are being reported [6] as they become more complex over time.

An alternative way to provide the monitors a kernel-independent execution environment is to introduce a physically isolated one. Compared to the hypervisor-based ones, these mechanisms have limited means to collect the kernel data or to interfere the kernel execution. On the other hand, they are less vulnerable to the software attacks in that their isolation does not rely on any software component.

This thesis follows the latter way to design and implement a kernel-independent monitor against the attacks on OS kernels. Before the mechanisms in this thesis being introduced, the monitors running on or rely on a physically isolated hardware component [7, 8] used periodically taken snapshots of the kernel memory to collect the information about the kernel. This not only introduces non-negligible performance overhead, but also is vulnerable against the *transient attacks* which remove their traces as soon as possible to avoid being detected by those monitors analyzing the periodic snapshots.

To address these two drawbacks of existing mechanisms, we propose a novel means for the physically isolated monitors to collect the kernel data accesses. Specifically, this thesis discusses the design and implementation of various *snoop-base kernel integrity monitors* [9, 10, 11, 12, 13] which use the novel means to monitor the kernel behavior to detect the attacks on OS kernels. Subsequently, we introduce another means which allows the monitors to acquire the kernel control-flow events, and present a mechanism using both to detect the attacks on OS kernels [14].

1.2 Hardware-assisted Enforcement of Data-Flow Integrity

As mentioned before, a number of techniques have been proposed to mitigate the memory corruption attacks. Among such techniques, the ones mitigating the violation of memory safety and Data-Flow Integrity (DFI) [15] are said to be the most comprehensive as many attacks violate both of them [1]. While more mechanisms, including the ones with architectural supports, have been proposed to mitigate the violation of memory safety, some programs cannot adopt them because they are designed to violate the memory safety intentionally [16]. This causes false positives when such a program adopts the memory safety enforcement mechanisms. On the other hand, DFI does not have such a problem but less number of mechanisms have been proposed and none of them were designed with the help of special hardware.

This thesis presents two architectural supports for mitigating the violation of *Data-Flow Integrity* (DFI) [15]. Though these are designed mainly for user-level programs, they could be adopted to harden the OS kernels also. The first technique is *Hardware-assisted Data-Flow Isolation* (HDFI) [17], which is designed to provide programs with isolated memory spaces where only the designated instructions can access. By letting only the instructions for the sensitive data use the isolated memory spaces, it is possible to isolate the data-flow of such data from the others. This makes it hard to corrupt those data by exploiting a vulnerability, as only the ones in the code blocks accessing the sensitive data data can be exploited to corrupt them.

The second technique is *Hardware-Assisted Randomization of Data* (HARD) that enables secure and efficient implementation of *Data Space Randomization* (DSR). Being an alternative way to mitigate the violation of DFI, DSR classifies the memory access instructions into equivalence classes such that any pair of instructions accessing the same object belong to the same class. With the classes, DSR instruments the program to encrypt the data before writing to the memory and decrypt them after reading, using the keys uniquely assigned to each equivalence class. Once a program adopts the DSR, any memory access that violates the DFI will use an incorrect key to encrypt or decrypt the data, resulting in unpredictable behavior of the program.

1.3 Outline

The rest of this thesis is composed of five chapters. Chapter 2 presents the design, implementation and use of snoop-based monitoring to mitigate the memory corruption attacks on OS kernels. The major part of the chapter has been published at *ACM*

Conference on Computer and Communication Security (CCS) [9] and *IEEE Transactions on Dependable and Secure Computing* (TDSC) [12]. Some parts of the chapter have also been published at *Usenix Security Symposium* [10], *Design, Automation and Test in Europe* (DATE) [13], and *Journal of Semiconductor Technology and Science* (JSTS) [11].

Chapter 3 shows the mechanisms to detect the code-injection attacks and code-reuse attacks on OS kernels with novel means of collecting kernel events. The major part of the chapter has been published at *Hardware and Architectural Support for Security and Privacy* (HASP) [14].

Chapter 4 shows the architectural supports for the data-flow isolation, which is a variation of the data-flow integrity. The contents of the chapter has been published at *IEEE Symposium on Security and Privacy* (Oakland) [17] and also included in the dissertation of its first author [18]. The author of this thesis made a significant contribution to the project mainly about the design and implementation of the hardware supports. For this reason, only the parts of the published paper about the hardware extensions are included. The ones about the security applications designed and implemented, and the security implications can be found in the published paper [17].

Chapter 5 presents the architectural supports for the data space randomization, which is a means to mitigate the violation of the data-flow integrity. Although this chapter has not been published yet, this was a highly collaborative work to which the author of this thesis made a major contribution.

The last chapter, Chapter 6, concludes this thesis.

Chapter 2

Snoop-Based Kernel Integrity Monitors

2.1 Motivations

To protect the integrity of operating system kernel against rootkits, many researchers strive to make their monitors isolated from the monitored host system. Recent efforts on this kernel integrity monitoring can be categorized into two groups: hardware based approaches [7, 19] and hypervisor based approaches [20, 21]. While approaches based on hypervisors have gained popularity recently, hypervisors themselves are exposed to numerous software vulnerabilities [6] as they are becoming more and more complex. Several approaches [8, 22] noted that inserting an additional software layer to protect the integrity of hypervisors may not be sufficient. The additional layer will introduce new sets of vulnerabilities in a similar fashion of the hypervisors; inserting another padding with a software layer for security may enhance security temporarily, but it does not provide a fundamental solution. As a solution for this, they introduced hardware-supported schemes to monitor the integrity of hypervisors.

Most of the existing hardware-based solutions to kernel integrity monitoring make use of snapshot analysis schemes; they are usually assisted by some types of hardware components that enable saving of the memory contents into a snapshot, and then perform an analysis to find the traces of a rootkit attack. Copilot [7], HyperSentry [8], and HyperCheck [22] are exemplary approaches on snapshot-based kernel integrity monitoring. A custom Peripheral Component Interconnect (PCI) card is used to create snapshots of the memory via Direct Memory Access (DMA) in Copilot, and the System Management Mode (SMM) [23] are utilized to implement the snapshot-based kernel integrity monitors in HyperCheck and HyperSentry.

Snapshot-based monitoring schemes in general have inherent weaknesses mainly because they only inspect the snapshots collected over a certain interval, missing the evanescent changes in between the intervals. Attackers can exploit this critical limitation of snapshot-based kernel integrity monitoring. If attackers know the existence and the interval of snapshot-taking, they could devise a stealthy malware that subverts the kernel only in between the snapshots and restores all modification back to normal by the time of the next snapshot interval. This is called as *scrubbing attack*, and HyperSentry [8] addressed this by making it impossible for the attackers to predict when the snapshots will be taken. However, attackers can still create a *transient attack* that leaves its traces as minimal as possible without knowing the exact time that snapshot is taken. If the traces are left in the memory for a short time, there is a chance that it can avoid being captured in snapshots and not detected, as the authors of HyperSentry were aware of. The term transient attack refers to attacks which do not leave persistent traces in memory contents, but it still achieves its goal by using only momentary and transitory manipulations. In addition, such snapshot-based schemes cannot prevent the modification of kernel code or data in the main memory. For this reason, they cannot find the original memory contents any place in the system, which is required to resotre the malicious memory contents to the genuine ones. Once the monitor detects any malicious modification, the only prescription for the compromised kernel is rebooting the system and exploiting the secure boot process [24, 25].

In this chapter, we propose Vigilare, a snoop-based integrity monitoring scheme that overcomes the limitations of existing hardware-based kernel integrity monitoring solutions. The Vigilare integrity monitoring system takes a fundamentally different approach; it monitors the operation of the host system by “snooping” the bus traffic of the host system from a separate independent system located outside the host system. This provides the Vigilare system with the capability to observe all host system activities such as writing to the main memory, before the activities take effect, and yet being completely independent from any potential compromise or attacks in the host system. This snoop-based architecture enables security monitoring of virtually all system activities, as all processor instructions and data transfers among I/O devices, memory, and processor must go through the system’s bus. By monitoring this critical path, Vigilare system acquires the capability to observe all activities to locate malicious system transactions, and prevent these transactions from being completed.

This Vigilare system is composed of the following components. *Snooper* on the Vigilare system is connected to the system bus of the main system, collects the contents of real-time bus traffic, and delivers the accrued bus traffic to *Verifier*. The main functionality of Verifier is to examine the snooped data to look for a single or a certain sequence of processor executions that violates the integrity of the host system.

In summary, this chapter’s contribution includes the following:

First, we present the prototypes of Vigilare system, each of which is designed to monitor a Linux kernel running on either the Gaisler grlib-based SoC or an ARM-based SoC. To the best of our knowledge, our Vigilare is the first hardware integrity monitor that implements the ability to snoop the bus traffic to perform an integrity analysis of an operating system kernel. Although most of the hardware-based approaches were based on snapshot analysis, one approach employed event-driven integrity monitoring [21], which has some similarities to snoop-based monitoring. However, it was a hypervisor-based approach which is not a perfectly safe execution environment as noted in the text.

Second, we propose an extended design of our prototype for the grlib-based SoC. This extension allows Vigilare system not only to detect malicious modifications to the host system main memory, which was supported by the first prototype, but also to prevent permanent damage to the contents of the main memory from malicious modifications. Although this design inevitably increases the memory access latency, we could minimize the performance overhead by exploiting the pipelined nature of the host system bus.

Third, to illustrate and provide a better understanding of the benefits of our snooper-based approach in comparison with previous snapshot-based approaches, we also implemented a snapshot-based monitor for the grlib-based SoC and created a sample transient attack testing code with tunable parameters for the experimental testing. The sample attack code takes advantage of snapshot-based solutions and we show how Vigilare effectively deals with these types of attacks.

Lastly, we present our comparison evaluation study, reporting that the prototypes of Vigilare system detected all of the transient attacks on immutable regions without degrading the performance, whereas the snapshot-based monitor could not detect all of the attacks and induced considerable performance degradation with an average of 17.5% for a 50ms snapshot interval in our tuned STREAM benchmark test. This is due to the performance overhead incurred by the memory access to acquire snapshots periodically, which also consumes memory bandwidth at the host. In contrast, the first prototype for the grlib-based SoC replicates the bus traffic of the host system using a dedicated hardware module such that snooping can be performed without consuming any memory bandwidth at the host system. The fifth prototype also incurs negligible performance overhead, as the increased latency can be hidden by pipelining. In addition, the prototype for the ARM-based SoC also successfully detected all transient attacks with negligible performance overhead, despite its weakness in computing power compared to its host system.

2.2 Assumptions and Threat Model

2.2.1 Assumptions

We assume that the host system is already compromised by an attacker, that is, the attacker has gained the administrator's privilege on the host system. In addition, we assume that the attacker has no physical access to the entire system therefore we can rule out the possibility of hardware modification. Thus, we are limiting the manipulation by the attacker in the realm of software; no modification should affect the operation of the Vigilare system hardware or any other hardware components.

2.2.2 Threat Model

The primary threat that the Vigilare system strives to mitigate is the kernel-level rootkits. As mentioned previously, we assume that the attackers have already gained control over the host system. In order to continuously steal data, and also to eavesdrop the users of the system, the attacker can conceal their footprints by manipulating the kernel, so that it can bypass the security mechanisms in the kernel or mislead the anti-malware solutions running on the host.

Another assumption is that the attackers might be aware of the presence of some kind of security monitor. The attackers aim to avoid detection with the best of their knowledge. One possible technique is to minimize and avoid an obvious and permanent modification to the kernel. By hiding the traces of malware left in the host memory, the attackers could lower the probability of getting detected. We define such malicious behavior transient attacks and the details will be covered in the next section.

2.3 Transient Attacks

As mentioned before, previous kernel integrity monitoring schemes that utilize memory snapshots to find the traces of rootkits are vulnerable to transient attacks. On the other

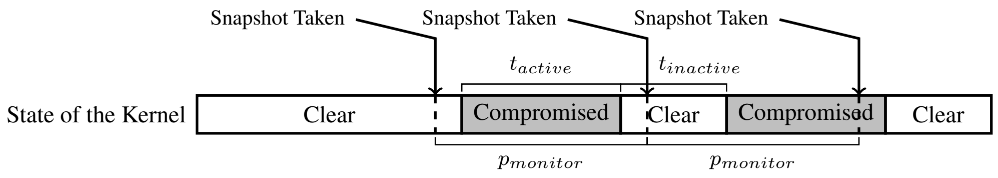

Figure 2.1: This figure shows the behavior of transient attack. Transient attack compromises the kernel "transiently" for t_{active} and removes its trace for $t_{inactive}$ to avoid being detected. Assuming that there exists a snapshot-based integrity monitor, it may detect the second pulse but fail to detect the first one. The transient attack with lower t_{active} may have more chance to attack the host system successfully.

hand, Vigilare solves such shortcomings with its bus traffic monitoring architecture. In this section, we define the term transient attack, introduce some examples of such attacks, and discuss the challenges in detecting these transient attacks.

2.3.1 Definition

Transient attack is an attack whose traces do not incur persistent changes to the victim system. In such scenarios, the evidence of malicious system modification is visible for a short time period. In turn, detecting the modification becomes difficult. The term transient attack is defined rather broadly; any attack that does not leave permanent changes can be classified as transient attacks. The soft-timer based rootkit technique presented in the work by J. Wei et al. shows the aspects of transient kernel rootkit [26]. The rootkit designed by them takes advantage of Linux *timer* data structure to convey the malicious code and execute it at the scheduled time. Since the code that had been contained in the timer is discarded shortly after its execution, its trace is not only difficult to locate but it also stays in the memory for a very short time. The exploitation of Linux soft-timer is a mere example, more sophisticated transient attacks that can nullify traditional rootkit detection solutions are here to stay.

2.3.2 Difficulties of Detecting Transient Attacks

In order to successfully detect transient attacks, the detection system needs to operate on an event-triggered mechanism; snapshot analysis or other periodic checks are likely to miss out the events in between the snapshots. Imagine a snapshot-based integrity monitor was launched to detect the transient attack model shown in Figure 2.1. If the author of the kernel rootkit can properly adjust the duration of the attack t_{active} and the time of dormancy $t_{inactive}$, he could completely evade the snapshot-based monitors; by staying dormant at the time of memory snapshot and becoming active in between the snapshots, the rootkit can capably fool the snapshot-based monitor.

A possible temporary solution to the limitation of the snapshot-based approach would be to increase the rate of memory snapshot-taking or to randomize the snapshot interval. Frequent memory snapshots will, however, inevitably impose a high performance overhead on the host system. Also, random snapshot timings may not properly represent the system status and will produce either snapshots with little differences or snapshots with a long time interval in between. Therefore, we conclude that snapshot-based integrity monitors are simply not apt for detection of transient attacks. That is, an event-triggered solution is essential to cope with transient kernel attacks.

2.4 Vigilare System Requirements

Vigilare is designed to collect the data stream in the host system's memory bus to overcome the limitations of memory-snapshot inspection. Figure 2.2 shows high level design of the Vigilare system. The Vigilare system is mainly composed of two components: Snooper and Verifier. Snooper is a hardware component which collects the traffic and transfers it to Verifier. Verifier is a compact computer system optimized to analyze the data in order to determine the integrity of the host system. In our prototype, Verifier is placed along with Snooper for simplicity and performance. In this section, we describe several requirements that the Vigilare system needs to meet, so that it does

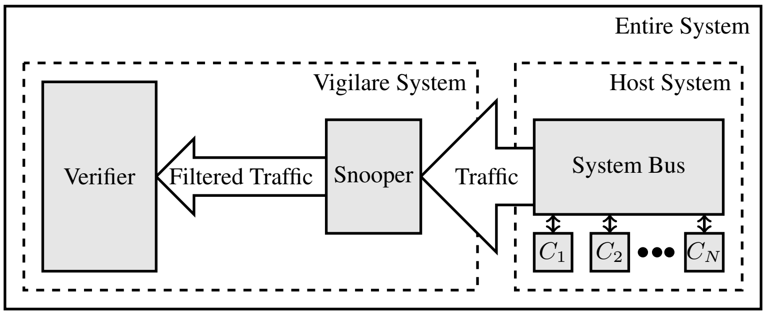

Figure 2.2: This is a high-level design of the Vigilare system. C_1 through C_N indicates the hardware components of the host system which are connected each other via the system bus, such as main processor, memory controller, or network interface. The Vigilare system includes a Snooper that has hardware connections to the system bus of the host system to be monitored. Verifier analyzes the filtered traffic that Snooper provides.

not fail to detect important kernel status changes that appear in the bus traffic.

2.4.1 Selective Bus-traffic Collection and Sufficient Computing Power

As Verifier in general needs to analyze the filtered collection of bus traffic that Snooper provides, a bus bandwidth that is higher than Vigilare's computing speed can be problematic. The AHB (Advanced High-performance Bus) included in AMBA 2 (Advanced Microcontroller Bus Architecture) [27], which was used in the host system of our first prototype is a good example of such a problem. Every cycle, 4 byte address and 4 byte data are transferred to memory controller through the bus along with a few more bits of bus-specific signals. Thus, we cannot process the stream of bus traffic in one cycle with a general purpose 32 bit machine. Therefore, Snooper must be designed with a selective bus-traffic collection algorithm; it should recognize only meaningful information while truncating other unnecessary traffic data flow.

The required time to process a filtered collection of traffic is also related to the computing power of Verifier. The more computing power Verifier has, the less time would be required to process the same collection of traffic. Verifier's computing power must be predetermined in the design process, so that it provides just enough processing power yet does not introduce excessive power consumption.

2.4.2 Handling Bursty Traffic

Just filtering out some traffic may not allow sufficient time to process all collections of traffic to Verifier. Filtering may reduce the processing load imposed on Verifier processor and enables Vigilare to cooperate with high bandwidth systems. However, even selective filtering does not guarantee that the rate of bus traffic collected is steady and expectable. That is, a deluge of meaningful information may coincidentally congregate within a short period of time, overwhelming the computing power of Verifier.

The most intuitive workaround would be to add a FIFO (first-in first-out) queue to Snooper to address the problem. Unfortunately, it does not effectively remedy the problem. First of all, implementing a FIFO for the specific architecture will inevitably bring up the hardware cost. In addition, it is rather difficult to estimate the proper size of the FIFO and we cannot afford to discard the critical information when the FIFO becomes full.

A better approach is to build a more abstract, interpretable data from the raw bus-traffic data in Snooper. It would require more logic implementation to make Snooper more complex. However, it would be much more efficient than simply increasing FIFO queue or equipping Verifier with powerful processor, since Snooper can filter and summarize the bus traffic so that the Verifier can avoid the handling of unrelated bursty traffic.

2.4.3 Integrity of the Vigilare System

Hardware-based integrity monitor has its advantage in the independence from the host system. Since it does not rely on the core functionalities of the host system kernel, the integrity of the host system does not affect the integrity of the Vigilare system.

To further augment this strength of Vigilare, the memory interface of the Vigilare system and interrupt handling of Verifier should be designed with considerations for independence.

The memory of the Vigilare system contains all the programs and data used by the Vigilare system. The host system must not be able to access this memory in any way. One way of fulfiling this requirement is the use of separate memory for the Vigilare system, which makes the Vigilare memory become physically tamper-free. Alternatively, we may implement a memory region controller that specifically drops all memory operation requests from the host system. This option may reduce the cost of hardware implementation compared to that of building a completely separate memory for Vigilare.

The interrupt handling in Verifier can be a factor that undermines the independence from the host system. Thus, any circumstances that might trigger interrupts to Verifier should be carefully designed. More specifically, the peripherals controlled by host system should have no or limited ways of introducing interrupts to the Vigilare system.

2.5 Detection of the Attacks on Immutable Regions

In this section, we describe three of our prototypes that we used snoop-based monitoring to detect the attack on immutable regions. After describing the immutable regions of Linux kernel which these prototypes monitor as background information, we describe the details of the three prototypes of the Vigilare system: *SnoopMon*, *SnoopMon-A* and *SnoopMon-S*, each of which investigates the integrity of immutable regions of the Linux kernel running on either the grlib-based system and the ARM-based system.

2.5.1 Immutable Regions of Linux Kernel

We define immutable regions as regions that are critical to the integrity of the operating system such that any modifications of the regions are deemed malicious. Protecting the integrity of immutable regions should be the highest priority, as any modifications of immutable regions in an attacker's favor would be the most detrimental type of modification to the system. The immutable kernel region constitutes critical components in the OS, and any compromise of this region would seriously affect all applications running on top of the OS. For instance, all countermeasures for detecting control-flow hijack attacks must be able to detect manipulations of immutable regions. Attackers can otherwise easily perform attacks by manipulating immutable regions. Therefore, our prototype of SnoopMon focuses on monitoring the immutable regions of the kernel.

As the target of integrity monitoring, we included kernel code region, system call table and *Interrupt Descriptor Table (IDT)*. Kernel code region is the most obvious example of immutable regions; the basic functionalities of the kernel must not be modified after the bootstrap, in other words, during the runtime. The system call table is another good example of immutable regions, as hijacking the kernel's system call often serves as an efficient way to control the kernel in the favor of an attacker. Modifying the system call table of the Linux kernel is a popular way to intercept the execution flow of the victimized system. The Linux system call table takes a form of an array of pointers. Each entry in the table points to a corresponding system calls such as `sys_read`, `sys_write`, and many more. The adversary could effortlessly hijack these system calls by inserting a function between the syscall table and the actual system call handlers. Most user mode applications as well as kernel mode ones, rely on the basic system calls to communicate with file system, networking, process information, and other functionalities. Therefore, taking control of the system call table enables one to control the entire kernel from the bottom. IDT is also an important immutable region as a critical gateway that kernel system calls pass through. By subverting such a low level system call invocation procedure, it is possible to hijack the system calls before even reaching the system call table.

2.5.2 Physical Addresses of Immutable Regions

As stated in [7], the virtual memory space used by the Linux generates semantic gap between the host system and the external monitor; independent monitors like Vigilare has no access to the paging files that manages the mapping between the virtual address space and physical memory address. Moreover, the operating system's paging mechanism often swaps out less frequently used pages to hard disk space.

However, the location in the kernel memory space in which the static region of kernel resides, is determined at boot time. Thus, we can reliably locate and monitor the important symbols within the static region. The virtual address of the kernel text is found in `/boot/System.map` file, which lists a numerous symbols used by the kernel; it is a look-up table that contains virtual addresses of symbol names. The symbol `_text` and `_etext` signifies the start and the end of the kernel's text section. The physical addresses of the text section can be calculated by adding the virtual addresses and a certain offset, which remain constant during runtime. The physical addresses of syscall table and IDT are also determined at compile time, and thus we can make use of them for monitoring.

2.5.3 SnoopMon

SnoopMon is the first prototype of the Vigilare system that is designed to monitor a Linux kernel for the grlib-based system. Specifically, it uses the Leon3 processor as the main processor which is a 32-bit processor [28] based on SPARC V8 architecture [29] provided by Gaisler. It is designed for embedded software with low complexity and low power consumption. The Leon3 processor has seven stage pipeline with Harvard architecture, 16KB instruction cache, 16KB data cache, and runs at 50MHz in our prototype SoC. The system uses 64MB SDRAM as a main memory and has some peripherals for debugging. It runs the Snapgear Linux [30] which is an embedded Linux customized for the processor and runs kernel version of 2.6.21.1.

Figure 2.3 shows the structure of SnoopMon, which is basically a separate computer

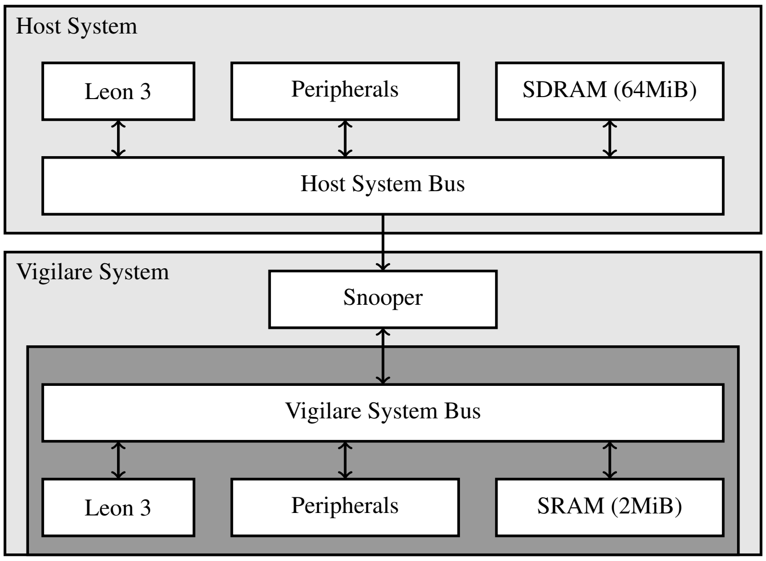

Figure 2.3: This diagram shows the architecture of our prototype with SnoopMon. SnoopMon has its own memory and peripherals for its independence from host system. It snoops host system bus traffic with Snooper. Snooper delivers traffic that indicates write attempts to immutable regions of host system kernel. The peripherals of SnoopMon include a debug interface for out-of-band reporting.

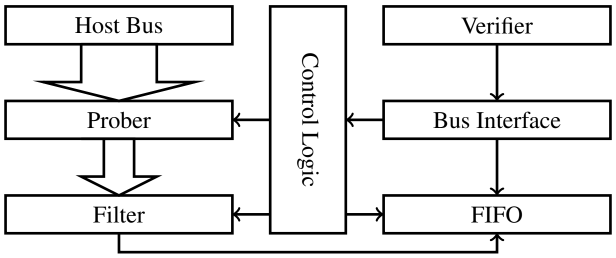

Figure 2.4: This is a detailed description of Snooper. Verifier controls Snooper with the AHB slave interface and control logic. Prober drops inessential signals in the incoming bus traffic to reduce the bandwidth of the remaining logics in Snooper. Filter is the key logic that compares the address field of the traffic with preset information pertaining to the immutable region.

system that consists of one Leon3 processor, Snooper, 2MB SRAM, several peripherals, and the bus that interconnects them. This configuration makes the memory and the memory interface of the Vigilare system independent from the host system. Moreover, the host system cannot access any peripheral of the Vigilare system and is also incapable of triggering interrupts to the Verifier. Therefore the Vigilare system design meets the requirement in Section 2.4.3.

To detect modifications to the immutable region that we explained in Section 2.5.1, SnoopMon specifically captures any write operation on those intervals of addresses, using its Snooper. As shown in Figure 2.4, the datapath of Snooper is composed of three modules: *prober*, *filter*, and *FIFO*. Prober drops inessential signals to reduce the bandwidth of remaining modules. AHB protocol defines many signals, but Vigilare system may not need all those signals for monitoring. For example, SnoopMon only needs HADDR and HWRITE signals among all the signals that AHB protocol defines. HADDR indicates target address of the traffic, and HWRITE is ‘1’ if and only if the traffic is to write. Filter is what determines whether to pass the traffic or not, and the filter for our SnoopMon compares HADDR with boundary addresses of the immutable region and checks whether HWRITE is ‘1’ or ‘0’. FIFO queue is the module to handle bursty traffic. FIFO queue stores the traffic that filter passed, and drop the traffic once Verifier pops it. It is worth noting that in this prototype, Verifier does not need to additionally examine the traffic from Snooper since only the write attempt to the immutable regions can pass the filtering of Snooper, all of which should be considered malicious. However, Verifiler of this prototype is still required as a kernel-independent entity that controls Snooper to monitor the immutable regions.

Although Snooper is capable of monitoring the bus traffic to the host main memory, access to memory contents that are in cache do not generate observable traffic on the bus. The bus traffic that indicates read attempts will be generated only when cache-miss occurs. On the other hand, write traffic is generated depending on the type of cache: write-through cache or write-back cache. In case of write-through cache, all write attempts by the processor may generate the corresponding packets on the bus. In case of write-back cache, the write attempts from the processor may not be seen immediately on the bus, because the write-back cache does not commit all the memory updates immediately to the memory. Thus, it is plausible to devise a transient attack that can live on write-back cache before the updated cache contents are flushed to the memory bus. However, predicting the time of the write-back cache flushing is not trivial, so implementing such a rootkit would be nearly impossible. In addition, attacks modifying the immutable regions cannot adopt the scheme to evade a snoop-based monitor. Even if an attacker successfully performed the attack, it generates dirty bits in cache lines of the regions, and it will be written back. Thus we can detect the attack by snooping the write traffic to the memory. Most caches write back dirty bits even when it is restored to original value, so any cache attack on immutable region generates corresponding write traffic on the links between caches and memories, where SnoopMon is watching. Simply enforcing write-through policy on the host caches would also be an alternative solution.

2.5.4 SnoopMon-A

We implemented another prototype of the Vigilare system in order to demonstrate more clearly the efficacy of our scheme in monitoring a host system that is either equipped with a write-back cache or run faster than the monitor. Using a development board with a *Zynq all programmable SoC* [31], we were able to implement SnoopMon-A, which monitors a Linux kernel 3.8.0 running on a system with an ARM Cortex-A9 processor as its main processor. As the host system is equipped with separate 32KB instruction and data caches as well as an 512KB write-back L2 cache, SnoopMon-A would show if our scheme suffers from the write-back caches or not. In addition, the processor operates at 666MHz and connected to an 512MB DDR2 SDRAM through a bus that operates at 120MHz, while SnoopMon-A operates at 20MHz and uses 16KB on-chip memory.

As SnoopMon-A operates at a frequency lower than that of the bus, adopting the design of the Snooper for the first prototype would cause the monitor to miss some traffic. In order to overcome the frequency difference, Snooper for SnoopMon-A has an advanced prober, which operates at the same frequency as the host bus and an asynchronous FIFO between the prober and the filter that bridges the clock domain. Unlike the one for SnoopMon, the prober for SnoopMon-A is designed to drop most of the traffic without hindering its detection capability by taking advantage of the line-based nature of memory accesses. As the host system uses a write-back cache, most write accesses from the host processor are generated with a granularity of cache line, which is 64Bytes in case of the host system. Consequently, snooping and processing the address of only one word among the eight words of a cache line, Snooper can infer the other seven addresses. Following this principle, the prober passes only one traffic among eight traffics if they are in the same cache line so that SnoopMon-A operating 20MHz can handle the traffic of the bus running at 120MHz.

2.5.5 SnoopMon-S

The third prototype of Vigilare system for the detection of the attacks on the immutable regions is SnoopMon-S. The earlier prototypes have been designed and implemented with the dedicated on-chip memory, but such a memory model is not desired in general due to the cost of the extra memory. In the cases that cannot provide the Vigilare system with dedicate memory, the only option is to use the memory for the host system. As mentioned in Section 2.4.3, the Vigilare system can still remain isolated from the host system, by introducing a memory region controller that prevents the host system from accessing the memory regions for the Vigilare system.

SnoopMon-S is implemented for the same host system that SnoopMon-A monitors, while it uses the protected regions in the host system memory with the help of the *region controller*. In fact, the region controller is merged to the Snooper because both modules examine the memory accesses and make decisions. Figure 2.5 shows the system with

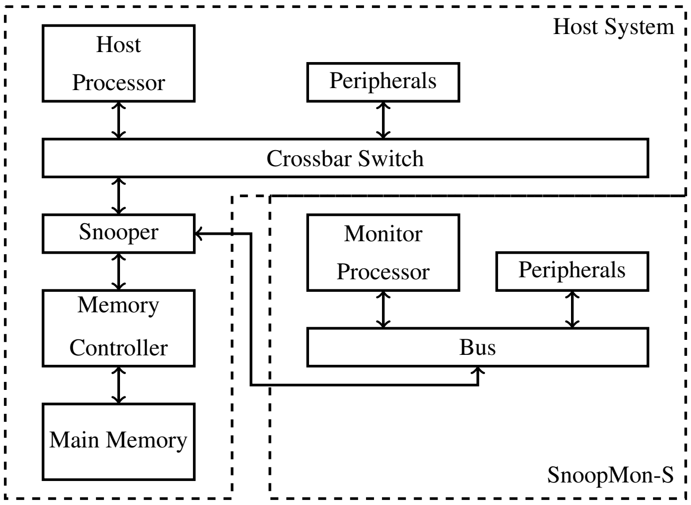

Figure 2.5: This diagram shows the architecture of our prototype with SnoopMon-S.

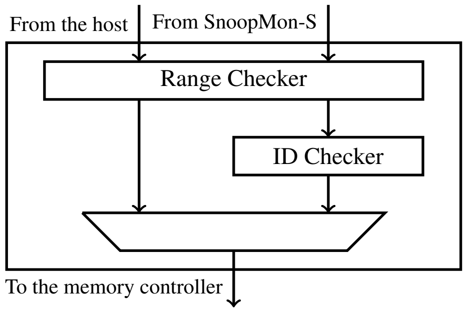

Figure 2.6: This diagram shows the structure of the region controller.

SnoopMon-S, and Figure 2.6 shows the structure of the region controller. Located in between the memory and the system interconnect, the region controller drops all memory operation requests from the host system to the memory region for SnoopMon-S. At first, the range checker module checks whether each memory access address is in the range of SnoopMon-S region. If it is not a transaction to the region for SnoopMon-S, the region controller passes it to the memory controller to access the host memory region. However, in the case of accesses to the SnoopMon-S region, the transactions from the host should be denied. For this purpose, the region controller discriminate the requests of the host with transaction ID of AXI protocol, being supported by the ID

Table 2.1: Examples of attacks on mutable kernel objects

Rootkit Name	Target Object Type	Object Type
Adore-NG 0.41	`inode->i_ops`	Control-flow component
	`task_struct-> {flags,uid,...}`	Data component
	`module->list`	Data component
Knark 2.4.3	`proc_dir_entry`	Control-flow Component
	`task_struct->flags`	Data component
	`module->list`	Data component
Kis 0.9	`proc_dir_entry`	Control-flow Component
	`tcp4_seq_fops`	Control-flow Component
	`module->list`	Data component
EnyeLKM 1.3	`module->list`	Data component

checker. Since AXI interconnection protocol supports transaction IDs to identify the master of the transaction, the ID checker can classify the master of the memory access and thus drop all the accesses from the host. By using the region controller, the memory region of SnoopMon-S is still physically temper-free and secure from the potential attacks of rootkits even in the unified memory model.

2.6 Detection of the Attacks on Mutable Regions

The attacks on OS kernels do not corrupt the immutable regions only. In fact, some rootkits in the wild are known to modify the *mutable regions* only, which results in avoiding the detection of the presented prototypes. To show the efficacy of snoop-based monitoring against the attacks on the mutable regions, we have developed another prototype of Vigilare system which can mitigate them. Specifically, the *KI-Mon* could detect two real-world rootkit attacks, the VFS hooking attack from *Adore-NG* and the *LKM hiding attack* from *EnyeLKM*.

The two examples that we choose represent real-world rootkit attacks on control-flow and data components. We analyzed the open source real-world rootkits [32, 33, 34, 35, 36] and referenced works that analyzed the behaviors of well-known rootkits [37, 38, 39, 40]. Table 2.1 summarizes some of the attacks on kernel mutable objects identified from the rootkits. These well-known rootkits manipulate both the control-flow and the data components. It is noticeable that the VFS hooking attack and its variants, which manipulates the control-flow components of Linux Virtual File System including the *proc* file system (VFS) [41], are popular for being deployed to hide files, processes, and network connections. Also, the LKM hiding was a common behavior among the analyzed rootkits. The attack manipulates a *module->list* structure to hide an entry in the *Loadable Kernel Module (LKM)* list. The rootkits utilize LKMs as a means to inject kernel-level code into the victimized kernel, and they launch the LKM hiding attack once their malicious code is loaded in the kernel memory space.

2.6.1 Attacks on Mutable Regions

2.6.1.1 VFS Hooking Attack

The Virtual File System (VFS) [41] provides an abstraction to accessing file systems in the Linux kernel; every file access is made through VFS in the modern Linux kernel. The kernel maintains a unique *inode* data structure for each file, which includes an *fops* data structure that stores pointers to the VFS operation functions such as open, close, read, write, and so forth. Various critical information about the kernel, such as the network connections and the system logs, are stored in the form of a file and are queried via the VFS interface.

Rootkits are capable of directly manipulating the functionalities of VFS. More specifically, they can hook the VFS operation functions of the fops data structure in a file to manipulate the contents read from it. Examples of malicious exploitation of VFS include hiding network connections or running processes, associated with the attacker. In Linux, /proc [41] contains important files that maintain system information. By hooking the VFS data structure that corresponds to /proc, the adversary can deceive administrative tools that rely on /proc for retrieving system information.

2.6.1.2 LKM Hiding Attack

Many rootkits take advantage of the Linux kernel's support of LKM. Initially designed to support extending of the kernel code during runtime without modifying and recompiling the entire kernel, LKMs often serve as a means to inject malicious code into the highest privilege level in a system. Moreover, adversaries often manipulate the linked list data structure that maintains the list of loaded LKMs in order to conceal malicious LKM loaded in the kernel. The following code line frequently appears in rootkits that are injected via LKMs:

```
list_del_init(&__this_module.list);
```

The kernel function `list_del_init` removes the given entry from the list in which it belongs. The developers of rootkits insert the code into the `module_init` function, so that the malicious LKM will be removed from the linked list upon its load. If the snapshot is not taken immediately, this attack cannot be detected because it removes itself from the linked list as soon as it gets loaded.

2.6.2 KI-Mon

The KI-Mon platform, including the monitored host system, is implemented as an *System on a Chip (SoC)* on an FPGA-based prototyping system for rapid prototyping. Figure 2.7 shows the overall structure of SoC with KI-Mon. The monitored system, running on a Leon3 [28] processor, configured to operate at 50 MHz. *Snapgear Linux* with a kernel version of 2.6.21.1 [30], provided from the provider of the Leon3 processor, was used as the operating system for the monitored system. Both KI-Mon and the host system use an AHB-compatible shared bus [27] as an interconnection network. As can be seen from Figure 2.7, the KI-Mon has been built on the same architecture base

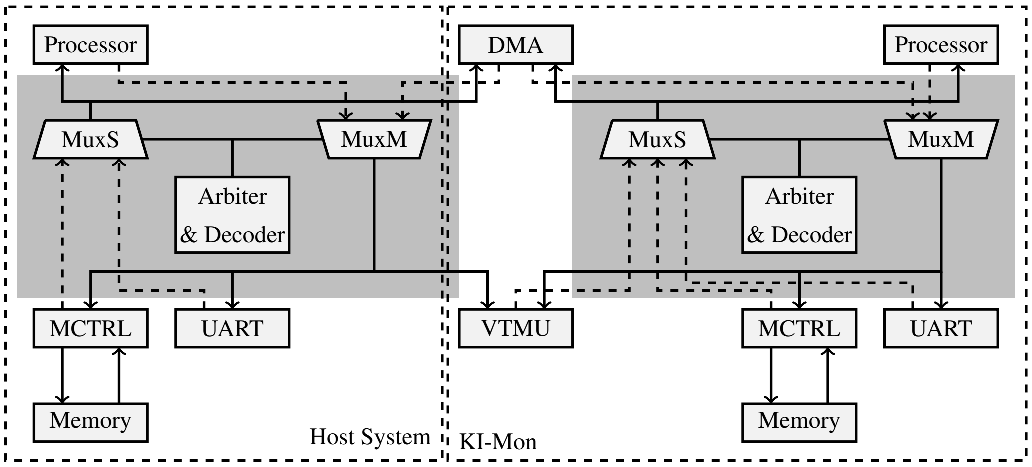

Figure 2.7: A system with KI-Mon

like SnoopMon as that of the host system, being augmented with new features for snoop-based monitoring.

To detect the attacks on the mutable regions, the data field as well as the address field of each memory transaction should be taken into account. For this reason, the Snooper for KI-Mon, which we call the *Value Table Management Unit* (VTMU), uses both the data and address to make decisions. By doing so, traffic that writes a known-good value to the monitored regions is ignored in addition to the traffic with addresses that do not belong to the monitored regions, as they are benign modifications.

The operation of VTMU consists of three stages: bus traffic snooping, address filtering, and value filtering. The first stage of VTMU operations, bus traffic snooping, is implemented based on a shared bus architecture that conforms to the AMBA 2 AHB protocol. Modules attached to the AMBA 2 AHB bus are categorized into masters and slaves. Masters are active modules that access slave modules as needed, whereas slaves are passive modules that respond to the requests of masters. In our implementation, the processor and DMA module are master modules, and the memory controller (MCTRL), serial port (UART), and VTMU are slave modules. The gray box in Figure 2.7 shows the bus architecture of the monitored system and the KI-Mon. Also, the connections

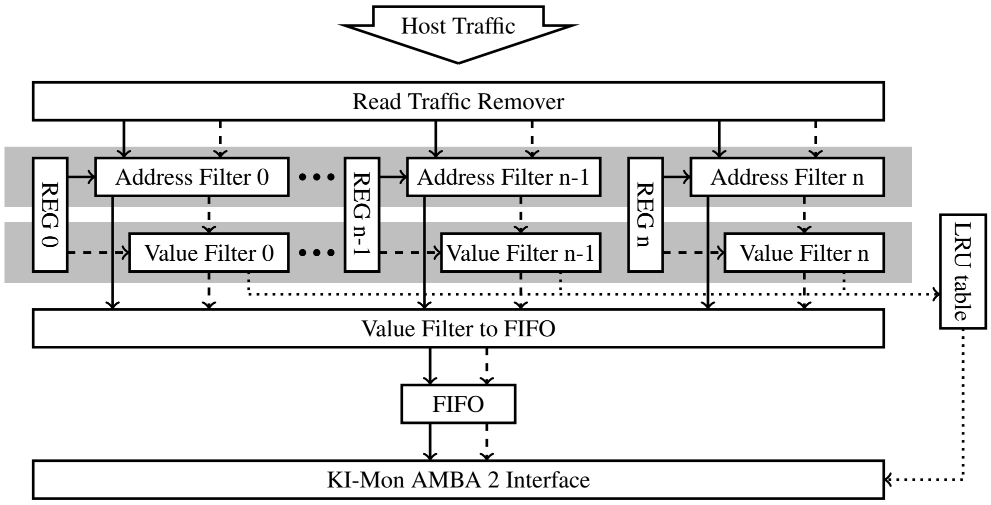

Figure 2.8: An overview of VTMU internal.

of VTMU on the KI-Mon are shown. *MuxM* is a multiplexer unit that passes only one master's traffic to a slave. MuxM is controlled by hardware logics called *arbiters* and *decoders*. These modules decide which master utilizes the bus at each clock cycle. That is, only one master can utilize the bus at each clock cycle, and all slaves receive the same traffic from the master at each time. With this hardware principle, we designed the bus traffic snooping stage of VTMU to acquire all memory traffic from the monitored system by duplicating the output signals of MuxM. The type of the traffic – whether the traffic indicates a write operation or not – is checked with a simple comparator, so that this stage only passes write traffic to the address filtering stage.

The overall view of VTMU's internal structure is illustrated in Figure 2.8. At the address filtering stage, the target address of each traffic is compared with the boundaries of the monitored region. Only the traffic to the monitored regions pass this stage, as any other traffic is benign and does not need to be checked. The traffic that has passed through this stage is fed into the value filters. The value of the traffic, or the value being written to the monitored regions, is compared with the values stored in the whitelist registers. If the traffic matches–meaning that this traffic indicates benign changes–it is discarded; if the traffic does not match, such bus traffic is stored in the FIFO buffer unit.

The FIFO buffer stores the output of the filter until that output is fetched by KI-Mon. Finally, VTMU notifies the Verifier of the address and value pair, generated from the FIFO buffer.

Other than VTMU, the hardware platform also includes a DMA module and a hash accelerator to support snapshot-related features. The DMA module takes snapshots of the monitored system's memory and stores them in KI-Mon's private memory. The DMA module has two master interfaces and one slave interface. One of the two master interfaces is connected to the monitored system's bus. The other is connected to KI-Mon's bus. With the master interfaces, the module is capable of reading any regions of the monitored system's memory; it can then copy the contents to the designated space in KI-Mon. The slave interface, which is connected to the KI-Mon bus, is used for the software platform to make requests for snapshots. The hash accelerator generates SHA-1 hash values from given memory contents. The hash accelerator has both slave and master interfaces to the KI-Mon bus. The slave interface is used to receive requests for hashing a certain region and returning the calculated hash value to KI-Mon, and the master interface is used to read the memory regions to be hashed.

2.6.3 Detection Mechanisms

2.6.3.1 VFS Hooking Attack

We observe that the VFS operation function pointers in the fops data structure store the addresses of the legitimate filesystem functions. For instance, the VFS function pointers of the data structure of a file in a *ext3* filesystem, point to ext3 operations in the kernel static region. In the same way, the fops data structure of a file in an *NTFS* file system includes pointers to NTFS operations. Using this property, we apply whitelisting to detect this particular attack. The procedural flow of the monitor is as follows: First, we trace the exact location of the fops data structure. Next, we set the function pointers as critical regions of the detection mechanism, and the location of the operation functions of the known file systems – such as ext3, ext2, and NTFS – as the whitelist. With these settings, VTMU notifies the software platform, which will subsequently provide a notification of this likely malicious event.

2.6.3.2 LKM Hiding Attack

By setting the `next` pointer of the LKM linked list head as the critical region of the detection mechanism, KI-Mon gets notified of the insertion of a new LKM as well as the address of the newly inserted `module` structure. When a new LKM is inserted, the software platform requests the DMA module to obtain a snapshot of the new module's code region and the hash accelerator to hash the contents of the region.

The rest of the procedure to verify if the new LKM is hidden from the list is as follows. First, the monitor waits for 30 milliseconds. Note that the wait time before this check is arbitrary. However, many rootkit LKMs include codes that hide the LKMs in the initialization function [32, 33, 34, 35]. Second, the linked list is traversed using the snapshot taken with the DMA to check if the inserted LKM is still in the list. Third, if the LKM is not found in the list, we walk the page table to verify that the virtual to physical address mapping that correspond to the LKM's code region has been deleted. The Linux kernel frees the memory regions of the LKM upon its removal. Therefore, the absence of the page table mapping to the region once occupied by the LKM indicates that the LKM was normally removed. In case mapping does exist, the last step of the procedure is executed. Recall that the monitor took a hash of the LKM's code region: we compare this hash against the hash of the current contents of the physical memory. If the two hashes match, this indicates that the LKM that was not found in the linked list iteration, is not properly freed from the memory. In other words, the inconsistency between the LKM linked list and the memory contents reveals the LKM hiding attack.

A page table consistency check is used to avoid the hash comparison of the memory contents, which requires additional processing time and memory bandwidth. The Linux kernel allocates the memory space for LKMs using `vmalloc` and de-allocates with `vfree`. The `vmalloc` function allocates a physically non-contiguous region of the requested size. That is, the allocated region is not necessarily contiguous in the physical memory, but is mapped to contiguous virtual addresses. Such non-linear mapping in the page table is deleted as the region is freed, using the `vfree` function. Therefore, the fact that the mapping is deleted in the page table assures that the LKM object is freed in the memory.

Even when page table mapping exists, it does not necessarily mean that a hidden LKM attack has occurred because the region that had been allocated for the LKM was possibly freed already and reallocated for another data object. Thus, a hash comparison of the region is necessary to verify the contents of the region. The kernel constantly allocates and de-allocates memory blocks from the non-contiguous memory regions for vmalloc requests. Therefore, it is likely that the freed region that used to hold a data structure object will soon be allocated for new one.

The consistency check is performed once, 30 milliseconds after the detection of a new LKM. This mechanism for the LKM hiding attack, is effective against known LKM hiding technique, deployed in many real-world rootkits. However, it is possible that rootkits evade the single fixed-timed check by delaying the execution of LKM hiding using a timer. To cope with such evasions, we can simply adjust the mechanism to schedule multiple random-interval checks for each occurrence of an LKM loading. For instance, we let the time of first check in seconds t_1 at the interval [0,5], the t_2 at [5,20], and so forth. By setting the lower bound of the random interval of t_n sufficiently long, we render the hiding attack ineffective; the longer the attacker has to wait, the effectiveness of the attack substantially diminishes.

2.7 Protection of the Kernel from Permanent Damage

In this section, we explain the motivation behind protecting the OS kernel from permanent damage and introduce the fifth prototype, which enables such protection by snooping and blocking bus traffic. We say that an OS kernel is permanently damaged if any malicious modification is committed to the main memory such that we can no longer find the clear kernel in the system. Upon the detection of malicious kernel modifications, Vigilare system needs to restore the kernel to a clear one as soon as possible. If this does not occur, the compromised kernel will continue to run as the attacker intended. A straightforward action of removing the effects of malicious modifications on the kernel is to reboot the kernel. If the host system provides a secure boot [24, 25] and the system has a set of untampered boot images, the reboot process always guarantees that the original, clean kernel image is loaded, thus repairing all the actions taken by the attack. This approach, initially, appears simple and effective, but simply rebooting the system without considering the consequences can entail several problems that should be noted. First, the system should have a kernel-independent means to protect the boot images from malicious modifications. As the attackers have the administrator's privilege, they are capable of modifying the images in the storage of the system. Although they cannot deceive the system to boot with a malicious boot image due to the secure boot, they still can destroy the boot image so that the system cannot reboot until the user somehow restore the boot image. In addition, an immediate reboot of the system could result in the loss of all unsaved data and statuses of all applications, as there are a number of applications running on the kernel at runtime. If this is not desirable, the reboot should be postponed until all the data and statuses are saved, which will allow the attackers to have longer period of time to achieve their goal. Thus, Vigilare needs to repair the kernel without rebooting.

We are able to achieve the goal by preventing attackers from writing to the system main memory. Once the kernel in the main memory becomes corrupted, there is no way to restore the compromised kernel to its original state, unless we duplicated all the previous values as backup. In contrast, any software-driven attacks can only corrupt kernel contents in the processor caches if the main memory is protected from such modifications. In this case, we can repair a compromised kernel in the processor caches by flushing the corresponding cache line. Because compromised cache lines are flushed

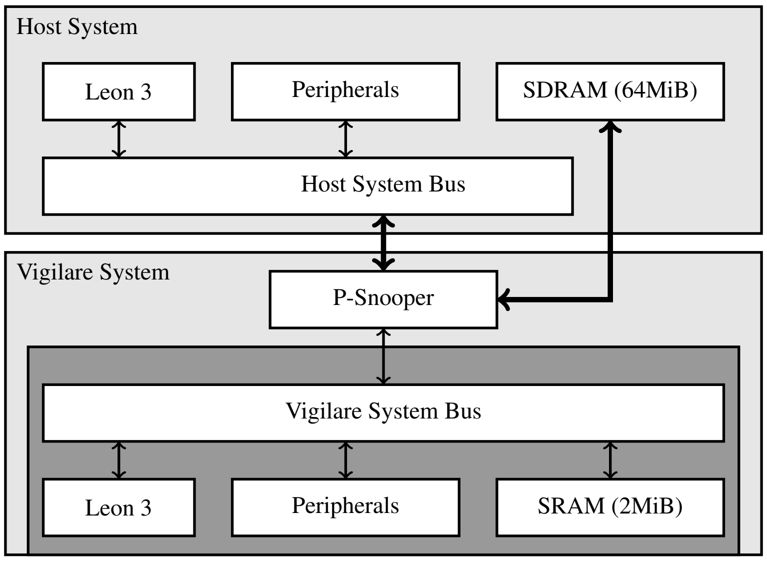

Figure 2.9: This shows the SoC including our fifth prototype of Vigilare System, P-SnoopMon. Unlike Snooper in SnoopMon, P-Snooper intercepts bus traffic between the host system bus and memory controller, and blocks malicious traffic.

without committing their changes to the main memory, the host system will continue to run a clear kernel in the main memory. Both the other prototypes and SnapMon cannot prevent this permanent damage to the kernel. SnapMon recognizes malicious modifications from a snapshot of the host system main memory after the kernel in the main memory is corrupted. The other prototypes may have a chance to detect a modification before the memory contents are changed, but it does not have any means of preventing the changes.

We designed and implemented P-SnoopMon, which not only detects all transient or persistent malicious modifications to kernel immutable regions as SnoopMon does, but also prevents all malicious modifications to the regions in the main memory of the host system. The latter allows P-SnoopMon to protect the host system kernel from permanent damage such that it can repair the compromised kernel without rebooting the system. For host systems with write-back caches, P-SnoopMon repairs compromised kernel image in the caches by flushing them and making use of the protected kernel

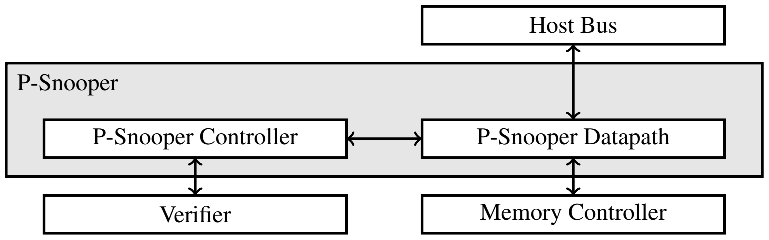

Figure 2.10: P-Snooper is composed of two components: P-Snooper Controller and P-Snooper Datapath. The latter passes write traffic to the memory controller only when P-Snooper Controller validates the traffic. P-Snooper controller includes the part of Snooper shown in Figure 2.4 except the prober of P-Snooper controller, which is connected to P-Snooper Datapath instead of the host system.

image in the main memory. For host systems with write-through caches, P-SnoopMon even prevents rootkit attacks on immutable regions without repairing.

Figures 2.9 and 2.10 show the architecture of P-SnoopMon, which is composed of Verifier and *P-Snooper*. Unlike SnoopMon, P-SnoopMon not only detects but also blocks write traffic between the host system bus and the main memory. When the host system bus sends write traffic to P-Snooper, P-Snooper Datapath blocks the traffic and waits until P-Snooper Controller validates the traffic, which takes one cycle. This capability allows P-SnoopMon to prevent kernel rootkit attacks from damaging the kernel permanently. In addition, as the host system of our SoC prototype has a write-through cache, the P-SnoopMon prototype prevents rootkit attacks on the immutable regions of the kernel, as described in Section 2.5.1. Although P-Snooper inevitably increases the memory access latency, this latency is much smaller than that of the main memory. We present a detailed discussion of the performance overhead in Section 2.8.3.2.

P-SnoopMon is also a separate computer system that is independent of the host system kernel. As shown in Figure 2.9, the detail of the Verifier is identical to that of SnoopMon. Although P-Snooper Datapath has an AHB slave interface which is connected to the host system bus, it is not controlled by the host system. P-Snooper controller, which is controlled by Verifier, controls the P-Snooper Datapath. P-SnoopMon Controller snoops bus traffic at P-Snooper Datapath to validate the traffic, and generate corresponding control signals for the P-Snooper Datapath. Given that P-Snooper is exclusively controlled by the Verifier and is independent of the host system, the P-Snooper design also meets the requirement described in Section 2.4.3.

2.8 Evaluation

In this section we present the results of our experiments using the two prototypes of the Vigilare system against the attacks on immutable regions (SnoopMon and P-SnoopMon) and the *SnapMon*. First, we compare the snoop-based monitoring and snapshot-based monitoring in terms of performance overhead and the efficacy in detecting transient attacks, using SnoopMon and P-SnoopMon. Then we provide our evaluation study on effectiveness of the snoop-based monitoring for a powerful host system and more attacks with SnoopMon-A.

2.8.1 Comparison with Snapshot-based Monitoring

To better compare our snoop-based monitoring scheme with the snapshot-based method, we implemented a snapshot-based integrity monitor called SnapMon, following the design of such monitors represented by Copilot [7]. From SnoopMon, we implemented the hardware of SnapMon by replacing the Snooper with a DMA-capable memory interface such that SnapMon uses DMA to get the snapshots of the immutable regions of host system's kernel periodically. SnapMon calculates the hash value for each kernel region snapshot, and compares it against the pre-calculated hash value of the unmodified immutable region. In order to implement the monitoring functionality on the hardware, we first tried to use the processor for hashing the contents and some memory spaces to store snapshots. The memory we used for SnoopMon was sufficient for SnapMon since

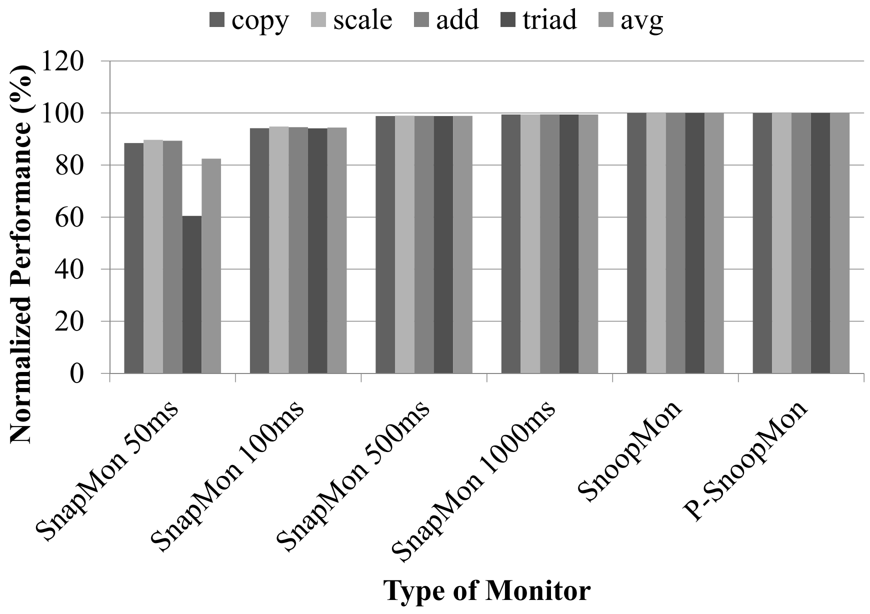

Figure 2.11: Performance degradation due to each monitor. Copy, scale, add, and trial are subbenchmarks of STREAM. SnapMon with shorter snapshot interval (i.e., increased snapshot frequency) degrades performance of the host more.

the size of immutable region in kernel was less than 1MB. However, the processing time for the hashing was longer than we had expected; it took about 5 seconds for each hash calculation for a snapshot. To reduce this, we included a hash accelerator in SnapMon to shorten the processing time to a reasonable level – approximately 1.3ms for each snapshot hashing.

We performed two experiments to compare the performance degradation and the efficacy in detecting transient attacks of each scheme. In case of SnapMon, the interval of snapshots significantly affects both performance degradation and ability to detect transient attacks. We varied SnapMon's interval to observe its effect on both detection rates and performance degradation. SnoopMon and P-SnoopMon do not have any parameter that possibly affects the result of experiments so we used only one type of SnoopMon and P-SnoopMon.

2.8.1.1 Performance Degradation

STREAM benchmark is widely used for measuring the memory bandwidth of a computer system. We used a tuned version of STREAM benchmark [42] to measure the performance degradations imposed on the host system. The original version uses double precision numbers for measuring the bandwidth but the Leon3 processor does not have floating point units. Thus, we tuned it to use integer numbers for measuring.

We let the tuned STREAM benchmark run on host system while each monitor is running. We averaged 1000 experiments to acquire more accurate results. As Figure 2.11 shows, SnoopMon and P-SnoopMon do not causes a discernible performance degradation, while SnapMon with shorter intervals degrades the performance significantly. For instance, performance degradation due to SnapMon with 50ms interval was measured to be 17.5% on average, 10% in the best case and about 40% in the worst case. If the snapshot interval is greater than 1 second, we have less performance degradation, 0.5% on average. This performance degradation was measured with the host system as-is after the boot, meaning that our benchmark tool is the only resource hungry process in the system. In addition, the performance degradation due to SnapMon also depends on the size of the immutable region, as it determines the size of snapshots to be taken and be transferred over. Note that both SnoopMon and SnapMon monitor the immutable regions that has the size of about 1 MB. As the Linux we used is tuned for embedded systems, the size of immutable regions may have been smaller than the kernels for desktops or modern smartphones.

2.8.1.2 Transient Attack

To compare the efficacy of the monitors in detecting transient attacks, we implemented a rootkit example that meets the definition of transient attack in Section 2.3. The rootkit modifies the system call function pointers in *sys_call_table* as traditional Linux kernel rootkits in the wild, but restores the table after t_{active}, to avoid being detected by a snapshot-based monitor, as shown in Figure 2.1. While the figure illustrates a transient

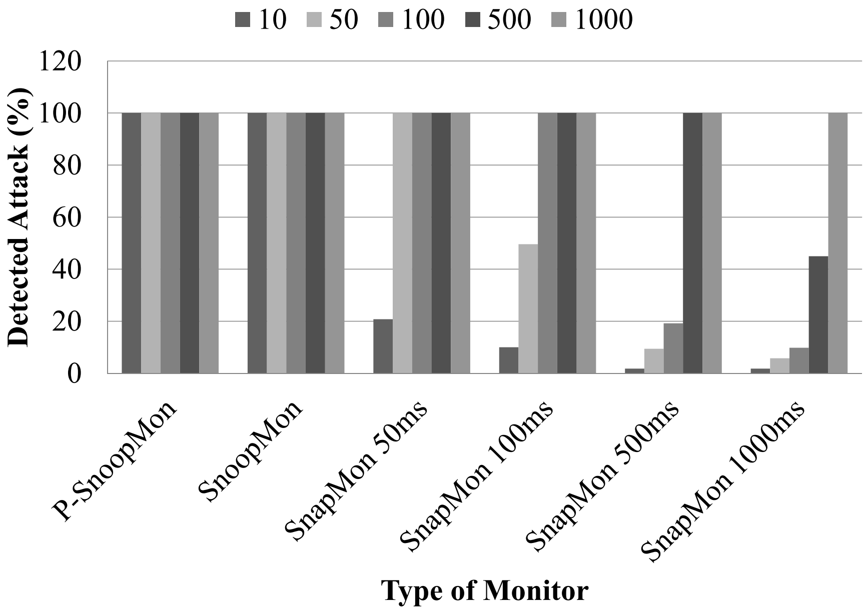

Figure 2.12: Ratio of detected attacks for each monitor. Numbers in legend indicates t_{active} of each pulse in the attacks in millisecond. This shows SnoopMon detects all the attacks while SnapMon cannot.

attack that performs hooking once again after $t_{inactive}$, our rootkit example has been designed to generate only one pulse of the attack so that we can measure the probability of detecting one pulse of the attack depending on t_{active}. Specifically, we generated 500 pulses and measured how many of them were detected by each monitor.

Figure 2.12 shows the results of the experiments. SnoopMon and P-SnoopMon detect all the pulses of attacks for all t_{active}, while SnapMon misses many of the transient attacks. Furthermore, P-SnoopMon also prevents all attacks that it detects, as mentioned in Section 2.7. SnapMon with 50ms snapshot interval detects all the pulses which have t_{active} greater than 50ms, but SnapMon with 1000ms snapshot interval cannot detect more than 5% of the pulses when t_{active} is less than 50ms. The results show that it is necessary to increase the snapshot frequency significantly for SnapMon to reliably detect transient attacks.

2.8.2 Effectiveness of Snoop-based Monitoring

In addition to the comparison study with the snapshot-based scheme, we have investigated the effectiveness of snoop-based monitoring through our additional prototype implemented on the ARM architecture, which we call SnoopMon-A. As we expected, SnoopMon-A caused negligible performance overhead regardless of the size of monitored regions, as well as successfully examined all write attempts to the immutable regions. We used the STREAM bench to measure the performance overhead, and checked if the asynchronous FIFO overflows or not, to make sure that Snooper receives every traffic representing a cache line eviction.

To further explore the efficacy of our scheme, we implemented three more rootkit examples in addition to the synthetic attack that we have described before, targeting the ARM-based system. One is another example of transient attack, which effectively hooks the system call table permanently for a particular process. The attack, which we call *per-process hooking*, manipulates the saved context information of the target process which is not currently running but in the scheduler's queue, in order to mislead the process to execute the *hooking payload* that performs system call hooking. To restore the system call table when the process is scheduled out, the payload also manipulates the kernel code such that the process invokes the *unhooking payload*, which restores both corrupted code and the system call table before switching to another process. This attack is transient in the sense that it only temporarily manipulates the immutable regions of the kernel, but effectively persistent to the target process as the hooking works for the process always. As this attack also modifies the immutable regions, SnoopMon-A could detect the attack when the target process is scheduled.

The other two rootkit examples are implemented to show the effectiveness of the snoop-based monitoring against the real-world rootkits, since all source code available rootkits that we have analyzed were designed for x86 systems. Because most of the rootkits have similar behavioral characteristic as shown in Table 2.2, we have implemented two rootkit examples for our host system following the representative

Table 2.2: *Key behaviors of ten Linux kernel rootkits.*

Name	Target Object	Object Type
Adore	sys_call_table	Immutable
Adore-NG 0.41	inode→i_ops	Mutable
	task_struct→{flags,uid,...}	Mutable
	module→list	Mutable
Knark 2.4.3	sys_call_table	Immutable
	proc_dir_entry	Mutable
	task_struct→flags	Mutable
	module→list	Mutable
KIS 0.9	sys_call_table	Immutable
	proc_dir_entry	Mutable
	tcp4_seq_fops	Mutable
	module→list	Mutable
EnyeLKM 1.3	sysenter_entry	Immutable
	module→list	Mutable
hideme.vfs	sys_getdents64	Immutable
	proc_root_operations	Mutable
override	sys_call_table	Immutable
Synapsys-0.4	sys_call_table	Immutable
fuuld	task_struct	Mutable
	sys_call_table	Immutable
net3	nf_hook_ops	Mutable

behavior of them. One manipulates kernel code region to intercept system calls, and the other modifies only mutable region to hijack the control-flow of the kernel. As we have expected, SnoopMon-A could detect the first example without difficulty, but not the second one as SnoopMon-A watches the modifications to the immutable regions only.

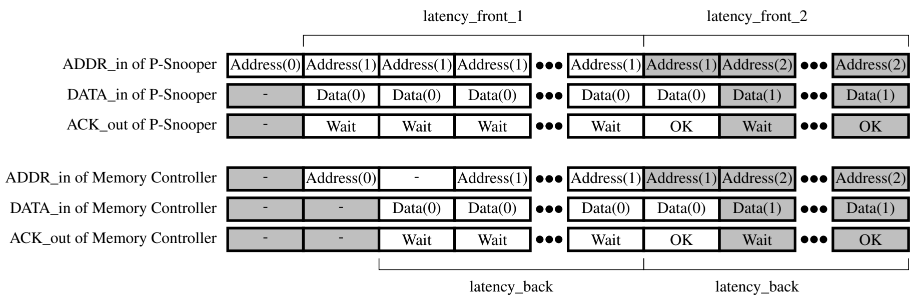

Figure 2.13: These timelines show the effect of P-SnoopMon on memory access latency. The bright part indicates the first transaction of consecutive memory accesses. Latency_-front_1 is the effective latency of the first memory access, and it is one cycle longer than latency_back, which would be the memory access latency of the host system without P-SnoopMon. However, the latencies of the other consecutive memory accesses are latency_front_2, which is the same latency_back

2.8.3 Discussions

2.8.3.1 Difference between SnapMon and SnoopMon

The comparison study shows that snapshot-based monitoring has trade-off between the performance degradation and the ability to detect transient attacks. With snapshot interval higher than 1 second, the SnapMon may cause little performance degradation but it misses out some transient attacks. Even the rootkit example with t_{active} 500ms, SnapMon with snapshot interval 1 second could only detect about 50% of them. In order to detect most of transient attacks, we need to lower the snapshot interval of monitoring but it results in significant performance degradation of the host system. However, SnoopMon and P-SnoopMon detect all the transient attacks with little or no performance degradation by employing snoop-based monitoring.

2.8.3.2 Performance Overhead of P-SnoopMon

As mentioned in Section 2.7, P-SnoopMon unavoidably increases the memory access latency, but the experimental results presented in Section 2.8.1.1 do not show any performance degradation. Figure 2.13 shows the reason why P-SnoopMon does not incur any visible performance overhead. Each block of the timelines in the figure indicates the abstract view of signals composing the traffic at each clock cycle. As shown at the beginning of the timelines, P-SnoopMon increases the memory access latency by one cycle, which is much smaller than the latency without P-snoopMon, latency_back in general. In our prototype, the value of latency_back is 12.

Furthermore, P-SnoopMon can pipeline a burst mode memory access by exploiting the pipelined nature of AHB protocol. As shown in Figure 2.13, the host system bus sends the second destination address, Address(1), before the first transaction is completed. If the latency_back is larger than 1, the additional memory latency due to P-SnoopMon can be hidden as shown in lower three timelines of the figure. P-SnoopMon can validate the next traffic while it waits for the response of the memory controller, since the value of latency_back is 12. Consequently, P-SnoopMon increases the latency of a burst mode memory access by only one cycle. Given that most of the memory accesses in the host system are performed in burst mode, P-SnoopMon causes negligible performance overhead as shown in the experimental result.

2.9 Limitations and Future Work

In this section, we discuss about some limitations, and future work.

2.9.1 Relocation Attack

Our snoop-based monitor prototypes cannot deal with relocation attacks. The relocation attack refers to moving the entire or parts of kernel to another location to stay out of the range of integrity monitoring. However, relocating, or copying of a large volume of the kernel code will inevitably produce an abnormal bus traffic pattern. Therefore, we expect that Vigilare could be extended to detect such attacks with a prior knowledge of the existence of such attack patterns.

This assumption varies across different computer architectures. If the system uses Harvard architecture, it would be relatively easier to detect the relocation attacks. For the initial relocation, the attacker should read the kernel code as data, not as instruction. Since such behavior is quite unusual, we can detect it by analyzing the traffic. Even for the systems based on the Von Neumann architecture, the traffic pattern on the immutable regions under relocation attacks would be far from normal status. In both cases, snapshot-based monitoring would not be helpful since it gets only an instance of the system state, not the sequence of actions changing the states.

Another type of a relocation attack, called *Address Translation Redirection Attack* (ATRA) [43], has recently been published. The monitoring of the page table pointing register is indispensable for attacks on address translations, and thus it would require a specific hardware modification. For this reason, we regard that a ATRA defense scheme would be out of the scope of this chapter. However, Addressing ATRA should be the most urgent future work for us.

2.9.2 Code Reuse Attacks

As the countermeasures against code injection attacks such as Data Execution Prevention (DEP) are employed in modern operating systems, many types of malware undertake a Code Reuse Attack (CRA) to exploit the vulnerabilities of target applications. CRA allows attackers to execute their own codes without injecting new code blocks or manipulating existing codes. Thus, CRAs work even when the execution of data and manipulation of codes are completely prevented. Rootkits can also perform CRAs to achieve their goals as shown in recent work [44, 45]. Although the proof-of-concept rootkit proposed in [45] overwrites system call table entries, which allows our prototypes to detect the rootkit, a conceivable rootkit may perform a CRA and avoid manipulating the immutable regions at the same time. Our prototype cannot detect this sort of attack, but countermeasures to these advanced attacks should also include solutions like our prototypes to mitigate simple but powerful attacks.

2.9.3 Privilege Escalation

While Vigilare concentrates on detection and prevention of kernel-level rootkits, the mitigation of *Privilege escalation attacks* are not discussed in this chapter. We consider privilege escalation attacks as a preliminary stage that leads to our attack model. The attacker executes an arbitrary code (by injecting or reusing existing) or have obtained the root user account. Then, the attacker subverts the kernel code with rootkits to track system activities or steal important credentials. Vigilare focuses on preventing this perpetuation stage of the intrusion by ensuring the integrity of the kernel static regions.

In addition, detecting privilege escalation attack is a rather broad topic. A privilege escalation might be achieved through a vulnerability in the code that is running with a kernel privilege (i.e., the ring 0), alternatively, a process running with a root privilege (i.e., UID=0) could also be exploited to concede the privilege to the attacker. Either way, the topic of privilege escalation attack mitigation would include nearly the entire realm of software attacks and mitigation techniques – the difference is that the targeted code is running with the privilege targeted by the attackers.

2.9.4 Cache Resident Attacks

Later we have been reported that one of the prototypes we have shown is vulnerable against the *cache resident attacks*. We define cache resident attacks to be the malicious attacks that, whether intentionally or unintentionally, take advantage of the fact the write-back caches do not evict the cache lines whenever they become dirty. The existence of caches on the host hardware can blindfold snoop-based monitors by impeding memory write events from appearing on the system bus, which consequently fertilize the environment for cache resident attacks of malware. Specifically, *loadable kernel*

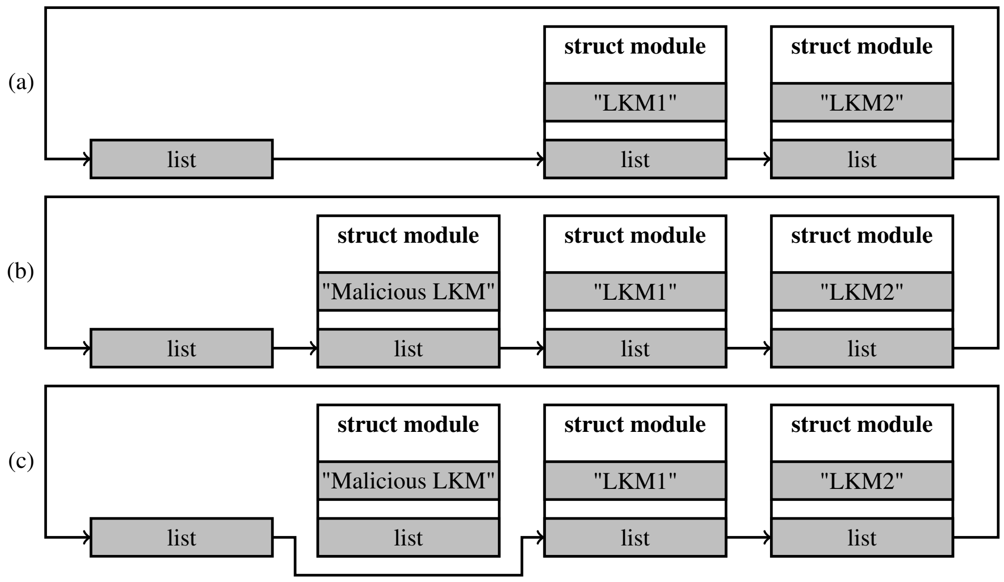

Figure 2.14: Cache resident attack example.

module (LKM) hiding can evolve to exploit this to avoid being detected by the KI-Mon.

Figure 2.14 shows how the LKM hiding technique is affected by write-back caches. As can be seen in (a), the kernel handles LKMs, each of which is represented by the `struct module`, by maintaining the `modules` list, which is a linked list of a set of `struct module`. The leftmost `list` is the head of the `modules` list, and it is statically allocated in the kernel data region. Upon the module load request, in this case a request from the malicious LKM depicted in (b), the kernel adds the corresponding `struct module` to the `list`. In a system with write-back cache, the `list` will be cached after this step, and subsequent accesses to the data structure will hit in the cache. Therefore, 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 will not be placed on the system bus.

This tendency of data being reside in caches can be exacerbated if attackers employ more aggressive techniques. One simple technique that can be readily applied is to regularly read the target kernel object using a kernel timer, affecting the decision of cache replacement policy of the host processor. Since recently proposed snoop-based monitors detect attacks by snooping the system bus, they might no longer guarantee the integrity of the kernel.

While it is not included in this thesis, a remedy to address this type of attacks have been proposed and successfully mitigated them [13].

2.10 Related Work

In this section, we explain previous approaches on protecting the integrity of an operating system kernel, which includes: kernel integrity monitors, rootkit detectors, and intrusion detection systems. The dilemma in designing such tools is that the security monitoring tool itself can be tampered with, if the malware operates on the same privilege level as that of the monitors. To cope with this problem, many security researchers strive to make their security monitors independent from the system that is being monitored. A separated and tamper-free execution environment must be preceded before any advanced detection scheme. [46]

We can categorize prior work that aims to provide a solution to the problem, into two groups: hardware-based approaches and hypervisor-based approaches. We summarize these approaches for the rest of this section.

2.10.1 Hypervisor-based Approaches

Virtualization solutions, commonly called VMMs (Virtual Machine Monitors) or Hypervisors are widely used nowadays to efficiently distribute computing power among different types of needs. Since the hypervisor resides in between the hardware and the virtual machines, the hypervisor possesses the scope to manage and monitor the virtualized operating systems.

There has been quite a few work that takes advantage of hypervisors for monitoring the security of the virtualized computers. One of the first in such work was

Livewire [20] proposed by Garfinkel et al. Livewire proposed security monitor installed virtual machines. More ideas based on hypervisor has been proposed and implemented on popular hypervisors such as Xen [47, 48].

Although positioned underneath and separated from the virtual machines, it has been warned that the hypervisors can be also exploited with software vulnerabilities. Many vulnerabilities of Xen are already reported and amended [6]. The discovery of hypervisor vulnerabilities might continue as the hypervisors are expanding in terms of code size and software complexity. This implies that the hypervisor might not be a safe independent execution environment, which is an imperative requirement for a security monitor.

There has been attempts to design minimal hypervisors for more secure execution environment for security monitoring [49, 50, 51, 52]. The idea is to include only essential software components to minimize the attack surface for software vulnerabilities. Some of such work used static analysis to ensure that their hypervisor is vulnerability-less.

Among such approaches, SecVisor [52] forces the CPU to execute only approved code in kernel mode by taking advantage of the Memory Management Unit (MMU) and the IO Memory Management Unit (IOMMU). This allows SecVisor to prevent un-approved codes from running in privileged mode, but limiting the use of mixed memory pages which contain both codes and data, which exist in numerous modern operating systems. It was necessary to modify the kernel linker script to remove the mixed code and data pages from the kernel.

In comparison with SecVisor, our prototypes cannot prevent maliciously injected code in data pages from being executed in privileged mode, but ours can monitor several important function pointers in immutable regions. Moreover, our prototypes can monitor an unmodified kernel regardless of the existence of mixed pages, and yet incur negligible performance overhead, whereas SecVisor cannot run an unmodified kernel and incurs greater performance degradation than the popular hypervisor software Xen.

Recently, Rhee et al. [21] proposed an event-driven integrity monitor based on hypervisor. With event-driven nature, it can be considered as a hypervisor version of snoop-based monitoring. However, the security of the integrity monitor itself heavily relies on the premise that the hypervisors are vulnerability free. Besides, it reported non-negligible performance degradation.

2.10.2 Hardware-based Approaches

Another approach in implementing a kernel integrity monitor out of operating system is attaching an independent hardware component. The idea of securing operating system using SMP (Symmetric Multi-Processor) was first proposed by Hollingworth et al. [53]. Later, X. Zhang et al. proposed IDS (Intrusion Detection System) based on a coprocessor independent from the main processor [19]. Petroni et al. designed and implemented Copilot [7], which is a kernel runtime integrity monitor operating on a coprocessor PCI-card. More snapshot-based work followed after Copilot and inherited the limitations of snapshot-based mechanism presented in Copilot [54, 55, 56].

Intel also contributed to the trend, by presenting a hardware-based rootkit detection called as DeepWatch [57]. J. Wang et al. designed HyperCheck [22] which is an integrity monitor for hypervisors based on a PCI card and the SMM (System Management Mode) [23]. A. M. Azab et al. also proposed a framework called HyperSentry [8] for monitoring the integrity of hypervisors with their agent planted in the SMM. The critical drawback of using SMM for security monitoring is that all system activities must halt upon entering SMM. It implies the host system has to stop, every time the integrity monitor on SMM runs. DeepWatch and HyperCheck focused on building a safe execution environment but they both utilized memory snapshots for integrity verification.

In all, most of the hardware-based approaches use memory or register snapshots [7, 22] as the source of system status information. However, they are inapt for monitoring instant changes occur in the host system and thus vulnerable to advanced attacks such as transient attacks. HyperSentry [8] also uses the state of host system at certain points of time, when the independent auditor stops the host system and execute the agent. Thus, this can be considered as a snapshot-based monitor along with Copilot [7] and HyperCheck [22], in the sense that they all use the periodically acquired status information. Our approach is fundamentally different from the previous snapshot-based approaches on hardware-based integrity monitors since our Vigilare is snoop-based monitor.

2.10.3 Snooping Bus Traffic

Snooping bus traffic is well known concept as shown in these two prior works. Clarke et al. [58] proposed to add special hardware between caches and external memories to monitor the integrity of external memory. The aim of this work is to ensure that the value read from an address is the same as the value last written to that address. It can defeat attacks to integrity of external memory, but cannot address rootkits nor monitor the integrity of operating system kernel, unlike Vigilare System.

BusMop [59] designed a snoop-based monitor which is similar to our SnoopMon, but the objective of BusMop is different from SnoopMon. BusMop is designed to monitor behavior of peripherals. Unlike BusMop, SnoopMon is to monitor the integrity of operating system kernel. To the best of our knowledge, Vigilare is the first snoop based approach to monitor OS kernel integrity while all of the previous approaches in this area were based on taking periodic snapshots.

2.11 Summary

In this chapter, we proposed snoop-based monitoring, a novel scheme for monitoring the integrity of operating system kernels. We investigated several requirements on implementing our scheme and designed the Vigilare system and its snoop-based monitoring. We focused on contributing improvements over the previous approaches in two main aspects: detecting transient attacks and minimizing performance degradation. To draw the contrast between Vigilare and snapshot-based integrity monitoring, we implemented SnapMon which represents snapshot-based architecture. We pointed out that the snapshot-based integrity monitors are inherently vulnerable against transient attacks and presented our Vigilare system as a solution. In our experiment, we demonstrated that SnoopMon and P-SnoopMon are capable of effectively coping with transient attacks that violate the integrity of the immutable regions of the kernel, while snapshot-based approach had their limitations. In addition, P-SnoopMon also protects the host system kernel against permanent damage. We also investigated the performance impact on the host system using STREAM benchmark [42], and showed that SnoopMon, due to its independent hardware module for bus snooping, imposes no performance degradation on the host. P-SnoopMon also caused negligible performance overhead although it increases memory access latency. Snapshot-based integrity monitoring proved to be unsuitable for detecting transient attacks in general; it is inefficient because of the trade-off between detection rates and performance degradation; higher snapshot frequencies might improve the detection rates, but the performance suffers from the overused memory bandwidth. In all, Vigilare overcomes the limitation of snapshot-based integrity monitors with snoop-based architecture.

Chapter 3

Protection of OS Kernels from Code-Injection and Code-Reuse Attacks

3.1 Motivations

These days, more and more attackers endeavor to compromise an OS kernel on which most of the applications in a system rely. Manipulating the kernel, attackers are capable of affecting the kernel's behavior in almost all aspects, such as the way how kernel objects are accessed, what data is sent through network, or what permission a file is accessed with. Unfortunately, like other programs, OS kernels could have vulnerabilities with which an adversary can acquire the capability to access their memory arbitrarily. For example, the Linux kernel, which is the kernel of the most dominant operating system in the smartphone market [2], is considered to have unknown vulnerabilities in that new ones are reported every year [3, 4]. Although they have been patched already, the adversaries would exploit a new one that is not published yet, to compromise a fully patched system.

A powerful way to compromise a victim kernel with the capability is the *code-injection attack*, so several mechanisms have already been proposed to detect it with architectural supports. With the access to the kernel memory, attackers can deceive the victim kernel into executing malicious code by placing it in a kernel memory region, corrupting some function pointers and manipulating the page table. To help OS kernels in mitigating the attack, modern processors are equipped with architectural supports, such as *Supervisor Mode Execution Prevention* (SMEP) [60] or *Privileged eXecute Never* (PXN) [61]. These supports add a field in page table entries, and make the Memory Management Unit (MMU) use the field to decide if an instruction can be executed with the kernel privilege or not. Utilizing these, several mechanisms have successfully detected the attacks by protecting the integrity of the page tables containing the configurations [62, 63, 64, 65, 66].

Although these mechanisms have successfully defeated the code injection attacks, they inevitably introduce non-negligible performance overhead. In order to detect the code-injection attacks with the new field in page table entries, they should mediate all updates to the page tables and ensure that only the pages with the legitimate kernel code are configured to be executable in the kernel mode. If they omit a single entry, an attacker can corrupt it to inject the malicious code into the kernel by marking it to be executable with kernel privilege and map the page to a physical memory region containing the malicious code.

In this chapter, we present Kargos, a hardware-based reference monitor that detects the code-injection attacks without mediating the accesses to the entire page table. Instead of marking each page with its permission, Kargos examines the target addresses of indirect branch instructions to detect the first control-flow transfer to a malicious code block, while the CPU runs in the kernel mode. In this way, the monitor can ensure that the kernel never executes with the kernel privilege the instructions from outside the predefined kernel code pages. In addition, Kargos checks if the virtual pages of the kernel code regions are mapped to the corresponding physical code regions correctly. Otherwise, the attacker would be able to remap the kernel code pages into a physical memory region filled with the malicious code [43]. Combining these two, the monitor can detect any execution of an instruction that is not fetched from the legitimate kernel code region while the CPU is in the kernel mode. For the sake of explanation throughout this chapter, we hereafter refer to the virtual pages storing the kernel code as the *virtual code regions* and the corresponding physical memory regions storing the kernel code as the *physical code regions*.

Even though Kargos needs to examine the target addresses of the indirect branch instructions, it is not necessary to install our Kargos inside the CPU core. Instead, our prototype is placed outside the CPU and acquires the values from the *Program Trace Interface* (PTI), which most modern CPU possess [67, 68]; in order to help debugging and profiling programs, the interface can be configured to continuously emit a stream of packets. Parsing them, Kargos can incessantly observe the target addresses of indirect branches.

Kargos can also secure the translations of the kernel code addresses without modifying the CPU core. CPUs usually use the values of some special registers and the contents of some page table entries for address translations. For instance, ARM processors have *Translation Table Base Registers* (TTBRs) that contain the base address of page global directory, and several entries of the directory are used for translating the kernel code pages. For this reason, it is enough to protect these values to ensure that the address of kernel code pages are translated correctly. To mediate the modifications to the special registers, we first add the instructions that check the correctness of the updated values, to the code blocks which update the registers. In addition, Kargos ensures that these code blocks are always executed as designed, by checking if they are executed atomically. Using the outputs of PTI, Kargos can enforce some code blocks, including the ones to update the special registers, to be executed atomically. At last, such code blocks can also be designed to notify Kargos of the new address of the page global directory. With the value, Kargos can always monitor every access to the directory entries for the kernel code pages with *bus snooping* mechanism [9, 69].

In addition to the detection of the code-injection attacks, these hardware supports for Kargos are also helpful to detect the attacks using the *Return-Oriented Programming*

(ROP). By chaining *gadgets*, which are instruction sequences already existing in the victim program, they can deceive the victim program into performing an arbitrary computation [70, 71]. As these strategy has also been shown to be effective for OS kernels [44, 45], some mechanisms have been proposed to defeat them [72, 73], but they were not designed for the Linux kernel which is widespread in use.

To date, the *shadow stack* is known to be an effective means to detect those attacks. If we collect the address of call site upon every function call and store it in a shadow stack protected from malicious modifications, it is possible to recognize any attack which corrupts the return addresses in the original stack. As the ROP attacks modify the return addresses in the original stack by definition, they cannot bypass this shadow stack-based detection mechanisms unless they nullify the mechanism itself. While such shadow stacks have been designed and implemented both with and without dedicated hardware support [74, 75], none of the existing mechanisms are suitable for the detection of the attacks on the kernel as they are not accompanied with the means to protect the shadow stack contents from the kernel-compromising attackers. Because it is possible to collect the execution traces and protect the shadow stack contents without relying on the OS kernel with the supports for Kargos, we applied these to the detection of ROP attacks.

To show the effectiveness and efficiency of Kargos, we have completely constructed its prototype in hardware to monitor Linux kernels that runs on an ARM-based system. According to our experiments, Kargos has caught all the kernel code injection attacks that we tested, yet incurring only about 1% performance overhead than the original system being left vulnerable to the attacks. The performance overhead of the ROP detection mechanism was also about 1% when tested with SPEC 2006 Benchmark suite. In addition, all components of our prototype are implemented in a physically secure hardware platform operating independently of the target system such that Kargos can work as planned even if we assume that an adversary is able to access any memory regions in the target system including the kernel code and page tables.

3.2 Problem Definition

This section describes the threat model and the assumptions for Kargos, in order to define the scope of this work.

3.2.1 Threat Model

In this work, we first consider adversaries who inject their code and hijack the kernel control-flow to execute the injected code in the privileged mode. The adversaries are hereby assumed to know of an OS kernel vulnerability (e.g., CVE-2014-3153 [3]) which they can exploit to access the victim's memory arbitrarily. With this capability, they are able to put their own code into the kernel memory and redirect a control-flow of the kernel to the code. The OS kernel may try to defeat the attack using architectural supports like PXN or SMEP, but such powerful adversaries can circumvent them by overwriting the page table entries, unless the entire page table is protected from such a corruption. In the application of Kargos for ROP detection, we also consider the attackers who try to avoid these mechanisms to detect the code-injection attacks by composing *gadget chains* to implement the code to be executed in the kernel mode [44, 45].

On the other hand, we assume that adversaries do not have physical access to a victim machine and the machine contains no malicious hardware. In other words, we rule out any kind of physical attacks as most previous work on kernel-independent security solutions do. On top of the assumptions about benign hardware, we add one more that our target system employs *secure boot* (or *trusted boot*) [24] to load the correct OS kernel image at bootstrap. Thanks to the secure boot, we assert that Kargos can safely collect, before the OS starts, the information about the kernel necessary for its monitoring job.

3.2.2 Assumptions

Kargos does not require the target system to have another privilege higher than the OS kernel for virtualization support, or the special CPU architecture for harboring the secure world. However, it still has some requirements. First of all, the target system CPU is assumed to have PTI, which in fact corresponds to the *program trace macrocell* (PTM) in ARM processors [67] or the *processor trace* in Intel x86 processors [68]. Luckily, modern processors today normally employ such hardware debug features, so we deem that this is a reasonable assumption. In addition, the target addresses coming out of PTI are assumed to be virtual addresses, which is indeed true for most PTIs in real machines. Lastly, the CPU may have a capability of controlling the interface but only by either executing some special instructions or accessing memory-mapped registers of the interface.

3.3 Code-Injection Attacks

To detect the kernel code injection attacks, Kargos watches the memory traffic and the PTI outputs in order to examine the memory accesses and the execution traces of the target system. In addition, the target system kernel is augmented to notify some events, such as special register updates or mode switches, and Kargos ensures that the augmented code blocks are executed correctly by checking if they are executed atomically or not. For the rest of this section, we first briefly describe the architectural supports which Kargos is equipped with, then more details will follow as to how Kargos detects the kernel code injection attacks with these supports.

3.3.1 Architectural Supports

As shown in Figure 3.1, Kargos is composed of two modules, *TraceMonitor* and *TrafficMonitor*. Each module has its own memory-mapped control interface connected to the CPU through the interconnect. At boot time, the kernel can initialize the modules

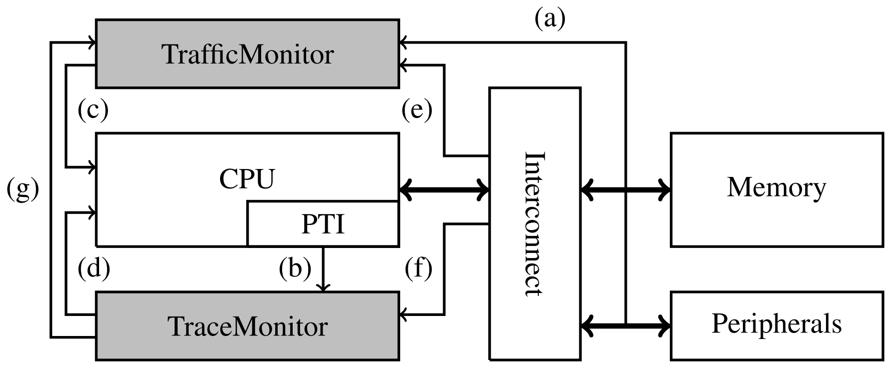

Figure 3.1: This figure shows how the hardware modules are connected to the other modules in the system. TrafficMonitor can examine the accesses to the memory or the memory mapped peripherals with the connections (a), and TraceMonitor is fed with the PTI packets through (b). Once one of them recognizes any violation, they interrupt the CPU to deal with the violation with (c) and (d). The CPU can access the hardware modules through (e) and (f), but the modules would accept these accesses selectively. TraceMonitor forwards some indirect branch target addresses to TrafficMonitor through (g).

through the interfaces, and can also pass various information such as the special register values during runtime. Nevertheless, no attacker can corrupt the configurations of the modules because each interface can be locked at boot time. Once locked, no software component running on the CPU can unlock them unless the entire system reboots. Because a reboot would make the system load a clean kernel image, an attacker would not be able to corrupt the configurations of the modules.

3.3.1.1 Trace Monitoring

The first architectural support is the *trace monitoring*, which enables Kargos to examine the indirect branch target addresses. Being originally designed for an external debugger to completely reconstruct the execution flow, PTI provides packets of information about the executeion trace. Specifically, an external debugger can calculate the target

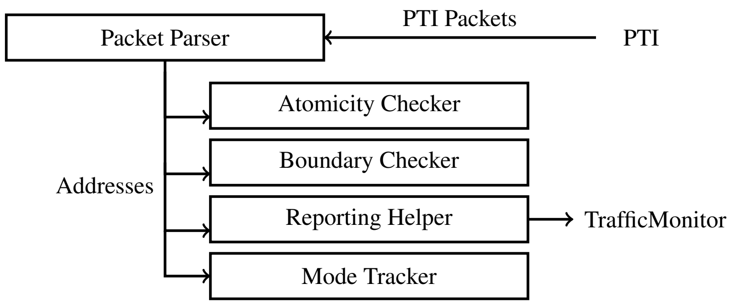

Figure 3.2: This figure shows the main components of TraceMonitor. The components that forward the addresses from PTI to TraffcMonitor and that generate an interrupt are omitted, and the functionality of each component is described in Section 3.3.1.1.

addresses of the indirect branches from the packets. As mentioned in Section 3.1, Kargos only needs the target addresses of indirect branch instructions, as the targets of direct branches can be statically analyzed and acquired. For this reason, TraceMonitor parses only the packets required to calculate the indirect branch target addresses.

Figure 3.2 shows the main components of TraceMonitor. The *Packet Parser* generates a stream of indirect branch target addresses from the outputs of PTI, and the other four components take this stream as an input and examine it. Among the components, *Boundary Checker* is what detects the jump to the malicious code while the CPU is in the kernel mode. It has a set of registers that contain the base addresses and bounds of the kernel code regions. By comparing the incoming indirect branch addresses with these values, TraceMonitor can determine if the CPU tries to run code fetched from outside the virtual code regions.

The second module is the *Mode Tracker*, which is mainly responsible for recognizing the mode switches. Located outside the CPU, neither TraceMonitor nor TrafficMonitor can directly access the CPU to acquire the mode information. For this reason, we implemented our system in a way that TraceMonitor can track the CPU mode switches using some special execution traces. More details about how we could generate such special execution traces securely will be presented in Section 3.3.2.2.

From the special traces, TraceMonitor can acquire the mode information securely.

The third module is *Reporting Helper*. To help TrafficMonitor to distinguish malicious *reports* of special register updates from the benign ones, this module selectively forwards some indirect branch target addresses to TrafficMonitor. In specific, the module forwards the target addresses of jumps to and from the code blocks which update the special registers for the address translations. Section 3.3.1.2 describes how TrafficMonitor uses these addresses to handle the incoming reports.

The last module of the TraceMonitor is *Atomicity Checker*, which enables the kernel to implement *Atomic Code Blocks*. Once atomic code blocks are realized in the kernel, this module is configured with the location information of the atomic code blocks, and checks if the CPU jumps to the middle of an atomic code block. Unlike the Boundary Checker, this module is not configured with the range of each code block mainly due to the scalability; the kernel can be built to be composed of a small number of code regions without difficulty, but could have an arbitrary number of atomic code blocks. For this reason, the Atomicity Checker uses (1) the ranges of regions containing the atomic code blocks and (2) the alignments to define the locations and boundaries of the code blocks. For each incoming indirect branch target address, Atomicity Checker first checks whether the target of the jump is in one of the memory pages containing the atomic code blocks or not. If it is a jump to an atomic code block, Atomicity Checker checks if the address is correctly aligned, to prevent the CPU from jumping to the middle of an atomic code block. For instance, if the size of atomic code blocks in the region is 256 bytes, Atomicity Checker would check whether the lower eight bits of the target address are zeros or not. In this way, the kernel can define a set of memory regions containing atomic code blocks and Atomicity Checker can ensure that the code blocks are executed atomically.

3.3.1.2 Traffic Monitoring

The second support for Kargos is the *traffic monitoring*, with which it can detect the malicious modifications to the kernel memory. Following the design presented in earlier work [9, 69], TrafficMonitor snoops the traffic between the *interconnect* and the memory, to detect any write accesses to the main memory, as depicted in Figure 3.1. Because any write accesses from the CPU to the caches make the corresponding cache line dirty and the dirty line will eventually be evicted to the memory, attackers cannot modify the memory contents without generating the corresponding traffic between the interconnect and the memory. In addition, because it is implemented in hardware, TrafficMonitor never misses any of the traffic. Note that TrafficMonitor can examine the memory accesses from the peripherals as well, including the Direct Memory Access (DMA), as there is no way to access the memory without using the link between the memory and the interconnect. Although this design may leave malicious modifications unrecognized when they are not written back to the main memory yet, attackers cannot take advantage of this to bypass Kargos or KS-Stack. Section 3.5.3.4 discusses why the attackers cannot take advantage of this.

To define the protected memory regions, TrafficMonitor has a set of base and bound registers. For each access to the main memory, TrafficMonitor uses the values of these registers to check if the access is a write to the protected region. For instance, the registers should contain the bases and bounds for the physical code regions to detect the malicious modifications to the kernel code including the atomic code blocks for special register updates, and those of the page global directory entries for the virtual code regions to protect them from being remapped. While the registers for the kernel code regions can be configured at boot time, the others should also be updated at runtime using the reports from the kernel because it changes the active page table by updating the corresponding special registers in the CPU.

In addition to the memory protection, TrafficMonitor is also capable of distinguishing fake reports from the genuine ones, which should be generated by a particular code block. Otherwise, the attackers would be able to generate a fake report to deceive Kargos into protecting a wrong memory region. As they are assumed to have an arbitrary access to the kernel memory, they may also be able to alter the values of the control registers in the devices. If so, such attackers would be able to deceive our system without difficulty, by accessing the control registers for the reports.

To ensure that a report is generated by the corresponding code block, TrafficMonitor generates a nonce which the code block should include as a part of the report. In addition, TrafficMonitor updates the corresponding registers with the incoming report only when it recognizes the execution of the corresponding code block, through the indirect branch target addresses from TraceMonitor. If the code block is designed to raise an alarm when it fails to fetch the nonce, attackers would not be able to deceive TrafficMonitor into updating the bases and bounds of the protected region with a fake report. Section 3.3.2.1 includes how Kargos uses this protocol and how the kernel implements a correct code block to generate the report.

3.3.2 Detection Mechanism

This section describes how Kargos detects the code-injection attacks using these architectural supports, beginning with the four rules that the system should comply with at runtime and that Kargos checks to detect the attacks.

R1. The physical code regions of the kernel should never be modified.

R2. The CPU jumps to an address in the virtual code regions when it enters privileged mode.

R3. All the targets of indirect jumps lie in the virtual code regions while the CPU is in privileged mode.

R4. All of the virtual code regions are mapped to the physical code regions. In other words, the CPU translates an address of a virtual code region into an address of a physical code region.

The target system CPU would not execute in privileged mode the injected code as long as it complies with these four rules. Owing to **R3**, if the CPU executes an instruction from the virtual code regions in privileged mode, the next one shall also be from the region. As it is possible to statically calculate all of the target addresses of direct branches and examine the instructions at the boundaries of the kernel image, we can ensure the following; if the CPU executes an instruction from the virtual code regions that is not an indirect branch, then the next instruction will be fetched from the virtual code regions. As the CPU must fetch the first instruction from the virtual code regions when entering privileged mode (**R2**), the CPU cannot execute any instruction from outside the virtual code regions when in privileged mode. Consequently, to execute injected code without violating **R2** and **R3**, an attacker should either make changes to the physical code regions to which the virtual ones are mapped, or remap a virtual page in the virtual code regions into an arbitrary physical page, which contains the injected code, that is outside the physical code regions. As the former violates **R1** and the latter does **R4**, we can conclude that it is not possible to perform a kernel code injection attack that complies with all the rules at the same time. Kargos can therefore recognize any sort of code-injection attacks if it is capable of detecting any attempt to breach the rules.

3.3.2.1 Securing the Kernel Entrance

To secure kernel entrances (**R2**), it is sufficient to ensure that the target system CPU jumps to the *gateway* code blocks whenever it enters privileged mode. Designed to be executed at that moment, these code blocks preserve the CPU states of the applications, examine the status registers to determine the reasons for mode switches, and establish the execution environment for the kernel. To calculate the addresses of the gateway code blocks, CPUs usually use the values of certain special registers, which we call *gateway registers*. For example, the Linux kernel for ARM CPUs has six gateway code blocks to handle system calls, interrupts and exceptions. To calculate the addresses of the blocks, the ARM CPUs use the values of the *system control register* (SCTLR) and the *vector base address register* (VBAR).

Kargos protects the gateway register values, which the attackers should modify in order to deceive the CPU into jumping to malicious code when it enters privileged mode. As the user applications should not be able to modify these values, the kernel in general has to execute certain special instructions, which can only be executed in privileged mode. To update the gateway registers of the ARM CPUs, for example, the kernel should execute the `MRC` instructions with some predefined operands [61]. If we can force the kernel to regulate all executions of the special instructions, attackers would not be able to corrupt the gateway registers. We augmented the kernel in a way that checks all the operands in the special instructions prior to its execution. This is done by executing the checking instructions before the execution of the special instructions.

However, the kernel cannot check all of the executions without additional hardware support, as it cannot prevent the CPUs from jumping to a particular address, especially to the special instructions. While the CPUs usually allow the kernels to disable exceptions or interrupts for atomic operations, they do not provide a means to prevent control-flow transfers to a particular set of addresses. If the kernel does not prevent these jumps to the special instructions, an attacker would be able to bypass the checking instructions and execute the special instructions, even if the checking instructions are directly followed by the special instruction. To deal with this type of conceivable attacks, Kargos uses the architectural support for the atomic code blocks, which is presented in Section 3.3.1.1. With the support, Kargos can guarantee that the special instructions always follow the checking instructions directly.

3.3.2.2 Detection of the Malicious Jumps

In order to recognize violations of **R3**, Kargos should be able to examine the indirect jumps. This is achieved with the help of TraceMonitor. At boot time, the kernel configures TraceMonitor with the ranges of the virtual code regions so that it can distinguish

```
msr      SPSR_fsxc, r1
and      r3, r1, #31
cmp      r3, #16
subeq    pc, pc, #4
restore_context
movs     pc, lr  ; this may switch the mode
```

Example 3.1: *This shows how the kernel is modified to generate a special control-flow in an exit code block. After updating the SPSR with the value in* `r1`*, we added three instructions to check the value in* `r1` *and generate an indirect jump if the mode field is set to the user mode.*

the jumps to these code regions from the ones to the other regions. However, TraceMonitor must also determine whether the CPU is in privileged mode or not so that it can ignores all jumps when the CPU runs in the user mode. As presented in Section 3.3.1.1, TraceMonitor is able to recognize the mode switches, if the kernel follows a special control flow when and only when the CPU enters or exits privileged mode. From such distinguishing traces of indirect jumps, Mode Tracker would be able to recognize the mode switches. Given that the kernel has already been enforced to execute the gateway code blocks when it enters the privilege mode, Mode Tracker considers the jumps to those code blocks as the signatures of mode switches from the user to privileged mode.

Similarly, OS kernels in general have several special code blocks that restore the CPU states of user applications and switch the CPU mode to the user mode. However, the jumps to these *exit* code blocks cannot be the signs of mode switches from privileged to user because the execution of the block may not always cause such a mode change. For example, the exit code blocks of the Linux kernel for ARM CPUs use `movs` instructions to switch the CPU mode to the user mode only when the mode field of the *saved program state register* (SPSR) is set to user. If not, the `movs` instruction does not change the CPU mode to user. For this reason, we augmented the kernel to follow a special control flow only when it returns to the user mode, whereas it follows the original control flow otherwise. In detail, the kernel checks the value of the SPSR before executing the `movs` instruction and executes an additional indirect jump instruction only when the mode field of the SPSR is set to user. This additional instruction generates a trace that Mode Tracker can consider as the sign of the mode change event. With this additional signature, Mode Tracker can recognize the mode switch from privileged to user. Example 3.1 shows how the signature for a Linux kernel running on an ARM processor is generated.

3.3.2.3 Protection of the Mappings

Although we could avoid protecting all page tables in the target system, Kargos has to protect the page table entries which map the virtual code regions into the physical code regions, due to the *Address Translation Redirection Attack* (ATRA) [43]. To protect these entries (**R4**), we make use of TrafficMonitor to examine every access to the entries. However, examining all these entries in the target system is not desirable as most systems maintain a set of such entries for each process. Instead, Kargos protects only the entries of the page tables currently in use, and lets the kernel check if the new entries becoming active and the old ones becoming inactive contain correct values to map the virtual code regions into the physical ones. In addition, we can also ensure that the kernel performs this checking whenever it changes the active page table using the hardware support for atomic code blocks. In the atomic code blocks that update the registers, the kernel also provides TrafficMonitor with the physical addresses of the new entries that should be protected from the modifications. In addition, the code block is implemented to comply with the protocol presented in Section 3.3.1.2 to be resilient to the fake reporting attack. Example 3.2 shows how the code block generates a report which TrafficMonitor would accept as a genuine one.

```
1    sub   pc, pc, #4
2    check_entries r0
3    bne   handler
4    ldr   r4, nonce_addr
5  wait_for_nonce:
6    ldr   r5, [r4]
7    cmp   r5, #0
8    beq   wait_for_nonce
9    str   r5, [r4]
10   str   r0, [r4]
11   dsb   sy
12   mcr   p15, 0, r1, c13, c0, 1
13   isb
14   mcr   p15, 0, r0, c2, c0, 0
15   isb
16   pop   {r4, r5, r6, r7}
17   orr   lr, lr, #0xc
18   bx    lr
```

Example 3.2: *This figure shows how the kernel follows the reporting protocol. Before checking the entries, the kernel executes an indirect branch instruction with which TrafficMonitor recognizes the execution of this block. If the checking operation fails, the kernel invokes a predefined handler at line 3. From line 6 through line 8, the kernel waits for a non-zero nonce to become available. Once it fetches a non-zero nonce, the kernel writes the nonce and the report the address of the new page global directory and the nonce to the corresponding control register (line 9 and 10), and execute the original code which updates the special registers about the page tables in use (line 11-15).*

3.3.2.4 Code Protection

To detect violations of **R1**, the kernel should examine all memory accesses to the physical code regions. Although kernels can rely on the MMU to examine the accesses and detect malicious modifications, this requires the kernel to have the means to protect the integrity of all page tables in the system. Otherwise, attackers would corrupt the page tables to deceive the MMU and modify the physical code regions without being

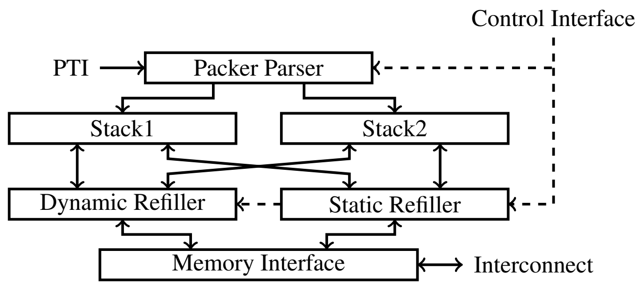

Figure 3.3: This figure presents the structure of KS-Stack. Section 3.4 explains the functionality of each component.

detected [62, 64]. Kargos does not require the entire page tables to be protected, as it can detect the write accesses to the physical code regions with their physical addresses, using TrafficMonitor.

3.4 ROP Attacks

On top of Kargos and the architectural supports for it, we have designed and implemented a mechanism to detect the ROP attacks with shadow stacks. As mentioned in Section 3.1, the existing hardware components for detecting code-injection attacks provide the Kargos with the two capabilities which are required for building and maintaining the shadow stacks. In this section, we describe the design of *KS-Stack*, which is an additional hardware component to detect the ROP attacks, and the software support for it.

As shown in Figure 3.3, KS-Stack is mainly composed of six modules, including two stacks. At runtime, only one of these stacks is activated; it is fed with the call site and return addresses by the Packet Parser of TraceMonitor. Having an interface to both of the stacks, *Dynamic Refiller* is responsible for managing the active stack by storing the contents into memory and loading the contents from the memory. To protect the contents of shadow stack staying in the memory, KS-Stack relies on TrafficMonitor to protect this memory region. As only one stack would be active at one time, this does not require the kernel to reserve a large amount of memory at boot time.

When KS-Stack receives information about the next process to be run, *Static Refiller* starts loading the corresponding shadow stack into the inactive stack, with hash checking. When KS-Stack receives an indirect branch address which indicates the invocation of the stack switching code block, it notifies both Refillers of the new state. Static Refiller then stores all of the contents of newly inactivated stack into the corresponding memory region.

Fed with the target addresses of all indirect branches, KS-Stack maintains the shadow stacks and examine the return addresses. While the implementation of a hardware-based shadow stack is not a new problem, we had to overcome two challenges due to our assumptions. First, our KS-Stack differs from the existing shadow stack implementations in that it cannot rely on any software components to manage and protect the contents of shadow stacks. In contrast, the existing shadow stacks can be managed by the OS kernel, as they are designed to protect user-level applications. In addition, KS-Stack uses only the address stream from the TraceMonitor to maintain the shadow stacks. Due to the limited high level information available included in the stream, it was necessary to augment the kernel to comply with the set of rules such that KS-Stack can infer the high level information from the target addresses.

```
1 .align  4
2   sub   pc, pc, #4
3   bic   r2, r2, #15
4   blx   r2
```

Example 3.3: *This shows an augmented indirect call site. The indirect jump at line 2 causes PTI to emit the call site address. To sanitize the indirect call, the kernel masks the function pointer at line 3. If this were a direct call site, line 3 should be replaced with a* `nop`*.*

3.4.1 Branch Address Classification

Connected to TraceMonitor, KS-Stack can collect all target addresses of indirect control-flow transfers, but it cannot distinguish the addresses of returns from the others due to a lack of high-level information in the trace. This is because PTI usually emits minimal required information, being designed to allow the debuggers to track the control flow of a target CPU. To enable KS-Stack to distinguish the addresses of returns from the others, we augmented the compiler and assembler to build the kernel code in a way that the *targets* of indirect branches are aligned according to the classes of the targets.

Like most programs, we can classify the targets of indirect control-flow transfers into three groups, which are the call sites, function entries, and the targets of in-function indirect jumps. With the augmented compiler and assembler, the targets in the kernel code are aligned such that KS-Stack can distinguish return addresses from others on the basis of their Least Significant Bits (LSBs). For instance, in our implementation, the LSBs of call sites are 1100b, and the others' LSBs are 0000b. Example 3.3 shows how we aligned the call sites. In addition, we insert an instruction before every return instruction to mask the corresponding return address in order to prevent an attacker from diverting the returns to an address which KS-Stack does not consider as a return. Without this masking step, attackers would be able to deceive KS-Stack by diverting a return to an address whose four LSBs are 0000b, e.g. a function entry. Line 17-18 in Example 3.2 show the masked return instruction. Relying on the aligned kernel and the masked returns, KS-Stack can recognize the addresses of return operations by checking the four LSBs of the addresses from PTI.

3.4.2 Call Site Emission

To maintain the shadow stack, KS-Stack should also be fed with the address of a call site whenever the kernel calls a function, but the address stream from TraceMonitor includes the targets of those calls only. For this reason, we augmented the kernel by inserting at every call site an additional indirect jump instruction before the corresponding call instruction so as to make PTI emit the *call site addresses*. For KS-Stack to distinguish such additional *call site jumps*, the targets of these call site jumps are aligned to have 1000b as their LSBs, as shown in Example 3.3. In doing so, KS-Stack can collect these call site addresses with LSBs 1000b and store the corresponding (calculated) return addresses in the shadow stack.

While we can ensure that the inserted call site jump instructions always change the program counter to the corresponding call instruction, an attacker may try to deceive KS-Stack by generating a fake call site address, by diverting one of the call instructions or indirect jump instructions. To prevent such an attack, we also mask the operands of call instructions and the in-function indirect jump instructions to have the correct LSBs, which are all 0000b in our implementation. With this alignment, it becomes safe for KS-Stack to use the LSBs to distinguish the call site addresses from the others. Line 3 in Example 3.3 shows an example of the masking instruction.

3.4.3 Protection of Shadow Stacks

As mentioned earlier, KS-Stack should manage and protect shadow stacks without relying on the OS kernel. Because OS kernels usually assign a stack for each process and there can be a number of processes invoked at runtime, it is inefficient for KS-Stack to manage all the corresponding shadow stacks concurrently. Therefore, KS-Stack should store the shadow stacks of inactive processes in memory and load one of them when the corresponding process becomes active. In order to fulfill these requirements, KS-Stack has a dedicated memory interface with which it can access the main memory without relying on the OS kernel.

In addition to the independent memory access, KS-Stack should also be capable of protecting the inactive shadow stacks in memory. Unlike the code regions or the predefined handler, the shadow stacks should be created and removed dynamically during runtime. For this reason, it is not ideal to reserve memory in which to store them at boot time and to use TrafficWatcher to protect it.

To minimize the amount of memory reserved for the shadow stacks, KS-Stack is equipped with a hardware component to hash the content of shadow stack. Using this component, KS-Stack can store the content where the OS kernel can access it arbitrarily, as KS-Stack would be able to detect any modification to the content by checking the hash value when it is loaded.

3.4.4 Context Switches

If KS-Stack manages one shadow stack to monitor the returns of a process as described above, it should be able to recognize context switches and change the active shadow stack correspondingly. Once KS-Stack knows which shadow stack it should switch to, it is not difficult to decide when to switch because most OS kernels invoke certain special functions to change the active stack. For example, Linux kernels call a special function, `__switch_to`, to change the active stack in use upon context switches. As KS-Stack can recognize this invocation, it can correctly determine the moment of the stack switch. To determine which one KS-Stack should activate after a context switch, we modified the kernel to maintain the location of the shadow stack for each process, and to notify KS-Stack of the location as soon as it determines the process which it switches to. Note that sending a false notification may not mislead KS-Stack to use an incorrect shadow stack, as it can verify the integrity of the shadow stack by means of the hash value.

3.4.5 Shadow Stack Creation

When the kernel creates a process, it allocates a memory block where the corresponding shadow stack should be stored. In addition, the kernel fills the block with the return addresses because the stack of a newly created process is not empty in general. When this process runs for the first time, the integrity of the corresponding shadow stack must be checked using a hash value, as described in 3.4.3. While KS-Stack does not fill the shadow stack of the newly created process, it can, at least in the case of Linux kernel, use a hash value precomputed at compile time to verify the content of such a

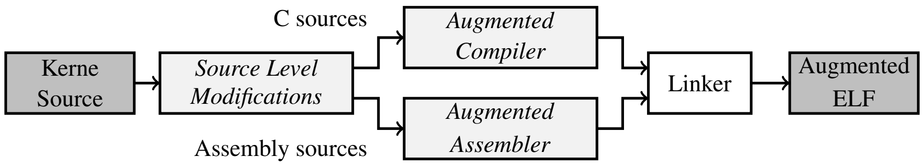

Figure 3.4: The procedure of kernel augmentation for Kargos and KS-Stack

shadow stack. Because all such processes should follow one of the predefined control flow before it returns to the user for the first time, we can compute the known good hash values of the newly created processes at compile time for KS-Stack.

3.5 Evaluation

To evaluate the effectiveness and the efficiency of Kargos, we have implemented a full-system prototype on the Xilinx ZC 702 evaluation board which includes Xilinx Zynq Z-7020 [31], on which an ARM-based system can be developed. the hardware modules are developed in Verilog HDL and mapped on the FPGA. Since the target system employs ARM NIC-301 AXI network interconnect, all the modules in Kargos are also designed to comply with the corresponding ARM AMBA 3.0 specification. Mainly due to the speed limit of FPGA, we configured Kargos to operate at 80 MHz, and also scaled down the clock speed of the target system to 222 MHz, complying with the performance ratio between the host and the coprocessors in most SoC platforms such as *application processors* of modern smartphones [76].

3.5.1 Implementation Details

On top of this SoC, we ran Android 4.2.2 with the Linux kernel 3.8.0 from the iVeia's git server [77] as the operating system. While we used the Android framework as it is, we modified the Linux kernel in order to implement the atomic code blocks for the special instructions. Specifically, we enclosed and relocated two types of instructions to secure the kernel entrances and the address translations, as presented in Section 3.3.2.1 and

Section 3.3.2.3, respectively. To secure the entrances, we created six atomic code blocks for the instructions modifying SCTLR. As the baseline system does not use VBAR to calculate the addresses, we did not need to protect it from malicious modifications. To protect the mappings, we enclosed four instructions modifying *Translation Table Base Register* (TTBR), which contains the base address of the page global directory in use. Because the kernel does not contain any special instructions for accessing PTM, we did not consider the case.

For the evaluation of KS-Stack, we used a different version of Linux kernel without Android on it, as we needed to build the Linux kernel with LLVM/Clang. On top of the same hardware configuration, we ran Linux kernel 3.17.0 which was built with LLVM/Clang 3.5 and GNU Binutils 2.23. Figure 3.4 shows how we built the augmented kernel ELF for our system. As well as modifying the kernel source directly, we used an augmented compiler and assembler to further augment the Linux kernel.

To maintain the location of the shadow stack for each process in the memory, we added a field in `task_struct`. When the kernel creates a new process, it allocates a memory region for the corresponding shadow stack, fills the region with the predefined content, as mentioned in Section 3.4.5, and sets the corresponding field in `task_-struct`.

In order to handle context switches, we added two code blocks to the `__schedule` function and the `__switch_to` function, respectively. As noted in Section 3.4.4, the kernel should inform KS-Stack of the location of the shadow stack to be used after the context switch as soon as possible. For this reason, we added the corresponding code block to the `__schedule` function, where the kernel selects which process to run. While KS-Stack can recognize the execution of the `__switch_to` function, KS-Stack may not have finished loading the shadow stack for the next process. We added a code block to make the kernel wait for KS-Stack to finish loading the shadow stack. When a process is killed, the kernel simply frees the corresponding shadow stack region.

Table 3.1: LMBench results. Numbers for Nested Kernel [64] is estimated since they are reported in the from of graphs only. As we could not find an existing mechanism that uses shadows stacks to detect the ROP attacks on OS kernels, we provide the numbers from KCoFI [72] as a reference.

Name	**Baseline**	**Kargos**		**Nested Kernel**	**Kargos+KS-Stack**		**KCoFI**
null syscall	0.98μs	0.97μs	(0.92%)	(0%)	1.04μs	(6.26%)	(142%)
open/close	18.39μs	18.15μs	(-1.28%)	(0%)	130.98μs	(68.5%)	(147%)
select	4.58μs	4.57μs	(-0.11%)	(-)	16.12μs	(33.74%)	(56%)
sig. handler install	2.81μs	2.82μs	(0.11%)	(0%)	2.91μs	(3.54%)	(114%)
sig. handler overhead	9.91μs	10.55μs	(6.42%)	(0%)	15.72μs	(58.66%)	(-8%)
pipe	40.89μs	43.23μs	(5.72%)	(-)	168.05μs	(66.43%)	(107%)
fork+exit	2853.15μs	2838.60μs	(-0.51%)	(190%)	13339.61μs	(17.05%)	(253%)
fork+execve	9279.8μs	9159.16μs	(-1.3%)	(150%)	110982.64μs	(18.35%)	(215%)
page fault	4.34μs	4.45μs	(3.63%)	(0%)	14.76μs	(9.82%)	(11%)
mmap	84.7μs	84.9μs	(0.24%)	(150%)	1121.19μs	(43.09%)	(229%)

3.5.2 Performance

To evaluate the performance overhead that Kargos and KS-Stack introduce, we initially used LMBench [78] to measure the performance of the operating system services. We used the script that is included in the benchmark suite to assess the performance impact on the latencies of the operating system services, as shown in Table 3.1. The reported values are the averages of 10 runs. As shown in the table, the performance impact of Kargos was negligible, even for the cases where Nested Kernel [64] suffer from considerable performance overhead. In addition, the impact of KS-Stack was moderate and much smaller than the performance overhead of KCoFI [72], an existing mechanism to detect the ROP attacks on OS kernels. It is worth noting that KCoFI may

Table 3.2: SPECint 2006 results. Among twelve benchmarks in SPECint 2006, our baseline system could not run 429.mcf due to the lack of memory.

Name	Baseline	Kargos	Kargos+KS-Stack
400.perlbench	12097.99s	12121.52s (0.19%)	12256.47s (1.31%)
401.bzip2	7284.54s	7274.29s (-0.14%)	7226.99s (-0.79%)
403.gcc	2420.82s	2429.91s (0.38%)	2416.94s (-0.16%)
445.gobmk	13412.38s	13542.57s (0.97%)	13437.86s (0.19%)
456.hmmer	15327.28s	15385.06s (0.38%)	15339.54s (0.08%)
458.sjeng	17000.11s	17051.94s (0.3%)	17056.21s (0.33%)
462.libquantum	42659.18s	42753.94s (0.22%)	42748.76s (0.21%)
464.h264ref	18785.86s	18841.65s (0.3%)	18847.85s (0.33%)
471.omnetpp	10334.19s	10382.46s (0.47%)	10316.62s (-0.17%)
473.astar	7717.71s	7684.35s (-0.43%)	7670.63s (-0.61%)
483.xalancbmk	11235.73s	11257.41s (0.19%)	11239.1s (0.03%)

detect some attacks that KS-Stack cannot [79], the opposite has recently shown to be true [80]. This makes it fair to compare the performance overhead of KS-Stack with that of KCoFI.

In addition to the microbenchmarks, LMBench, we also ran SPECint 2006 to evaluate the impact of our scheme on user-level applications. As shown in Table 3.2, the performance impacts of both Kargos and KS-Stack on SPEC are negligible. This suggests that our schemes would not degrade CPU-intensive workloads running as user-level applications.

Lastly, we ran five real-world Android benchmarking applications as shown in Table 3.3. All the results presented here represent the average values over 10 runs. As can be seen in the table, Kargos places less than 1% computational loads on average upon the target system, thanks to our architectural supports. As TZ-RKP [62] had to

Table 3.3: Android benchmark results. We could compares these numbers with TZ-RKP [62], as it used a similer set of benchmarks to report their performance numbers.

Name	**Baseline**	**Kargos**	**TZ-RKP**
RL	607.90	610.82 (0.48%)	(1.51%)
CF-Bench	531.80	527.80 (0.75%)	(7.55%)
GeekBench	67.20	67.00 (0.30%)	(6.80%)
Linpack-single	9.01	8.96 (0.64%)	(0.85%)
Vellamo-metal	121.80	121.40 (0.30%)	(7.65%)

Table 3.4: Postmark results. We could compare our results with the one presented in KCoFI [72].

Name	**Baseline**	**Kargos**	**Kargos+KS-Stack**	**KCoFI**
Total	480s	484s (0.8%)	546s (13.75%)	(96%)

protect the entire page table to detect the code-injection attacks, it inevitably incurred more performance overhead, as the table shows.

As we could not run the Android on top of the kernel built with the LLVM/Clang, we also ran another application benchmark to see the performance impact of KS-Stack as well as Kargos. For this purpose, we ran PostMark [81]. We configured PostMark to use 5000 file with sizes ranging from 500 bytes to 900 bytes, and to perform 10000 transactions. Table 3.4 shows that the execution time increases about 14% when our scheme is applied. This overhead was again smaller than the one reported in an earlier work [72].

3.5.3 Security

To evaluate Kargos and KS-Stack in terms of security guarantees that they provide, we tested Kargos with three Proof-of-Concept (PoC) attacks, and analyzed KS-Stack to reason that it can detect the ROP attacks as designed.

3.5.3.1 Detection of Code-Injection Attacks

As described in Section 3.2, we consider an attacker who exploits a vulnerability of the kernel to acquire the capability of accessing the victim system memory arbitrarily in order to perform the kernel code injection attacks. For subverting our target system, such attackers can exploit a real-world vulnerability of Linux kernel, which has been reported as CVE-2014-3153 [3]. To test the effectiveness of Kargos, we wrote three PoC attacks exploiting the vulnerability because we could not find any publically available kernel code injection attack on ARM-based systems leveraging the vulnerability. The first attack aims to execute its own code through modifying the physical memory regions, and the second one writes to its own page table to remap the virtual code regions. As both of them generate write attempts to the memory regions that the TrafficMonitor is monitoring, Kargos was able to detect the attacks without difficulty. The last attack is designed to hijack the kernel execution flow to already-injected code without modifying the kernel code regions or the page tables for the virtual code regions. As the hijacking inevitably causes a jump to the addresses outside the virtual code regions, the TraceMonitor raises an alarm when it receives the trace that corresponds to the jump from the PTI.

It is worth noting that the only difference between our PoC attacks and the publically available examples of kernel code-injection attacks on *packetstormsecurity.com*, which were used for the security evaluations in some earlier works [13, 10, 69], are the way how they achieve the capability of accessing the kernel memory. While the public examples are implemented as kernel modules to manipulate the kernel code and data, ours exploit a real-world vulnerability of the kernel to achieve the capability.

3.5.3.2 Detection of ROP Attacks

In order to make the kernel execute a gadget chain, attackers should cause at least one of the return instructions to change the program counter intentionally. Due to the masking, attackers can divert the function returns only to an address that has the special LSBs for return targets. As a result, any control-flow transfer to an address that is not in the shadow stack would be detected by KS-Stack. To the best of our knowledge, *Counterfeit Object-oriented Programming* (COOP) [82] is the only technique with which attackers can execute a gadget-chain under this restriction. We discuss COOP further in Section 3.6. Note that the indirect branch instructions we added use only the `PC` value, which attackers cannot control, as demonstrated using the examples in Section 3.4. This makes it impossible for attackers to abuse the inserted instructions.

3.5.3.3 Attacks on Mode Switches

Attackers who are aware of the Kargos may try to mislead the hardware modules by providing the incorrect mode of the CPU. If, somehow attackers successfully falsify the hardware modules, e.g. notify the module that the CPU is in user mode while the real CPU is not, they might be able to divert the control flow arbitrarily because all indirect branch addresses generated by executions in user mode are ignored. However, such an attack can be nullified because the gateway registers are protected from malicious modifications; thus, they cannot cause the CPU enters privileged mode without letting the hardware modules recognize the event. Due to the special control flow in exit code blocks, they cannot delude the hardware modules into believing that the CPU has left privileged mode. Note that the opposite, deceiving the modules into believing that the CPU enters privileged mode while the CPU is in user mode, does not help them to achieve their goal, though it might be possible.

3.5.3.4 Cache Attacks

As mentioned in Section 3.3.1.2, attackers may try to bypass Kargos or KS-Stack taking advantage of the timing gap between the malicious memory access and its detection. As processors would perform memory accesses to the caches but TrafficMonitor can recognize them only when they are written back to the main memory, there is a non-zero timing gap between the malicious memory access and its detection. In addition, it is known that there is a case where an attacker can bypass a reference monitor which relies on the memory traffic between the caches and the main memory [13]. We argue that this type of attack does not enable the attackers to bypass the mechanisms proposed in this work.

For the two mechanisms, TrafficMonitor watches every memory access to three memory regions. The first one is the physical code regions, which contain the kernel code. As these regions are static and immutable at runtime, any write access to these regions will raise an alarm. In other words, attackers cannot adopt the strategy shown in the earlier work about the cache attacks [13], as TrafficMonitor does not need the entire history of the memory updates to distinguish benign memory accesses from the malicious ones.

The second region TrafficMonitor watches is the region containing the active shadow stack contents. While the contents of an active shadow stack are stored in the hardware stack, they are written back if the stack becomes too long to be stored in the hardware stack. As these cannot be hashed due to their dynamic nature, KS-Stack uses TrafficMonitor to watch this memory region. Unlike the other regions, the processor does not need to access the region of these shadow stack contents. In addition, even if an attacker tries to corrupt this memory region, such attack will not take effect until the updated values are written back, i.e. the attack is detected.

The last regions are the ones for the page table entries mapping virtual code regions. Although these regions are also immutable while the entries are in use, they are immutable only a certain period of time, as TrafficMonitor monitors only the entries in use. As mentioned in Section 3.3.2.3, Kargos ensures that the page table entries have the predefined values when the entries become active. As this is checked by an atomic code block inside the OS kernel, this step ensures that both the entries in the caches and those in the memory contains the correct value. This makes it impossible for the attackers to corrupt the inactive entries.

To evade the verification step, the attackers should modify the page tables in use in order to redirect the virtual code pages into the physical pages with their malicious code. Due to the timing gap, however, the corresponding memory traffic may not be generated before the entries become inactive and TrafficMonitor stops watching the entries. This will also be detected by the checking code in the kernel, which is forced to be executed using the atomic code blocks. As the processor runs the code to check the integrity of the page table entries becoming inactive, the code can recognize the modified entries even if they are not written back to the main memory yet.

3.6 Limitations and Future Work

3.6.1 Bypassing the Scheme

While KS-Stack limits the capability of attackers considerably together with Kargos, it cannot prevent all control-flow attacks on OS kernels. For instance, attackers can perform a code-reuse attack that stitches functions rather than short code blocks at the end of functions, such as COOP [82], as our implementations allow the kernel to call any address-taken function at any indirect call sites. Although we could not find an example of such an attack on OS kernels, and COOP is designed for programs written in an object-oriented language, it might be possible to devise a similar technique to attack an OS kernel. Adopting fine-grained CFI [83, 73] or context-sensitive CFI [84] on our scheme will help securing the system by further hardening the kernel.

3.6.2 Kernel Modules

Modern kernels are allowed to load the *kernel modules* to extend their code at runtime. Although the current design of Kargos assumes that the code regions remain unchanged, the design can be enhanced to allow the target kernel to load the modules. In addition, it would be possible for the enhanced Kargos to decline an attempt to extend the kernel with a maliciously crafted module as long as Kargos has a set of cryptographic hashes of the known good modules. In order to load a kernel module, the kernel should send a request to Kargos to adjust the current virtual code regions, physical code regions and page tables. Otherwise the execution of the extended code will raise an alarm. Upon receiving the request, Kargos will firstly include the module image to the physical code regions to detect the attempt to modify the image. Consequently, the image becomes neither writable nor executable during the verification process through the hash. Once the module is confirmed to be a known-good one, Kargos then checks the image again to find if it contains the special instructions. After this verification process, Kargos can safely extend the virtual code regions to permit the CPU to execute the kernel module.

3.7 Related Work

3.7.1 Page Table Protection

As mentioned in Section 3.1, several mechanisms [62, 63, 64, 65, 66] have been proposed to protect the page tables utilizing the recent hardware supports such as PXN or SMEP. By protecting the page tables, they could detect the kernel code injection attacks effectively, but they inevitably introduced non-negligible performance overhead. Compared to these mechanisms, Kargos can defeat the attacks with smaller performance overhead, by avoiding the protection of the entire pare tables.

3.7.2 Hypervisor-based Approaches

With the higher privilege than the OS kernels, hypervisors in general are capable of investigating the kernel memory and intervening the kernel events. Using these capabilities, many mechanisms have been proposed to protect the kernel with the hypervisors. Among them, SecVisor [52] and NICKLE [85] are closely related to our work in that their main goals are also to detect kernel code injections. SecVisor protected and augmented the page tables whenever the CPU enters or exits the kernel to simulate the PXN/SMEP, as those supports were not available then. NICKLE emulated the *Havard Architecture*, where the code memory and data memory are strictly separated, by intervening the instruction fetches. Due to the lack of hardware supports, these mechanisms have higher performance overhead when compared with Kargos or the recent mechanisms using PXN/SMEP.

3.7.3 Snapshot Analyses

Another direction of the mechanisms to protect the kernel has been the *Snapshot Analysis*. Using either a dedicated hardware or hypervisor, it is possible to implement a monitor which acquires the snapshots of the kernel memory periodically and analyzes them to detect the anomalies. The earlier ones have focused on protecting the kernel code from being corrupted [7], but recent ones have checked the *Control-Flow Integrity* (CFI) [40, 47], by examining the function pointers in the kernel memory. These mechanisms would be able to detect the code-injection attacks if they corrupt the function pointers, but very recent work [86] suggested that a *dynamic hook*, which hijacks the execution flow without persistently altering any control data, would successfully evade these control data detection schemes. Although OSck [40] has a potential to detect dynamic hooks, they did not present a specific scheme that can handle all kinds of dynamic hooks on the kernel.

3.7.4 Bus Snooping

As mentioned in Section 3.3.1.2, the idea of bus snooping is not new and already proposed as a means to watch the accesses to the kernel memory [9, 69, 10]. However, they have focused only on the detection of the malicious modifications to the kernel memory, and could not detect the code-injection attacks effectively. In particular, they could detect code injection attacks that corrupt the physical code regions or some function pointers that they are monitoring. Unlike these mechanisms, Kargos can detect any type of kernel code injection attacks.

3.7.5 Control-Flow Integrity for Privileged Software Layer

Similar to KS-Stack, HyperSafe [87] and KCoFI [72] enforce CFI on the most privileged software layer without introducing an additional layer. However, their mechanisms are still vulnerable to return-oriented attacks, as they are not equipped with a shadow stack. On the other hand, our scheme is robust against these attacks because it employs shadow stacks to examine every return address.

3.7.6 Software Diversification

Another line of research has attempted to mitigate the threats by randomizing the code layout [88], thereby giving attackers little chance of finding gadgets for their attacks. Giuffrida et al. [89] randomized the layout of the kernel code so that attackers cannot compose a gadget chain unless they disclose the code layout. Moreover, they suggested periodic re-randomization of the layout to fortify their scheme against memory disclosure attacks [90]. However, their randomization mechanism is mainly designed for microkernel-based operating systems. Therefore, as they also mentioned in their work, their scheme would incur higher performance overhead when applied to monolithic kernels. Our scheme, on the other hand, is designed for and implemented on monolithic kernels such as Linux. Recently, Gionta [91] presented a mechanism that randomizes the kernel code and restrains the code region such that it is only executable. However, their method relies on a hypervisor to implement the execute-only memory.

3.7.7 Formally Verified Microkernels

Klein et al. [92] presented seL4, a formally verified microkernel, and proved various properties of the kernel [93, 94, 95, 96]. As they noted in their paper, formal verification is the only way to guarantee that a system does not have vulnerabilities, thereby leaving no entrance for attackers into the system. Under such an environment, attackers cannot arbitrarily access the kernel memory; thus, they may not be able to undertake control-flow attacks on verified kernels. Unfortunately, however, most systems today employ Windows or Linux as their OS kernels, which are not formally verified. In the current situation, we believe that our hardening scheme may be able to help these OS kernels efficiently to defend against control-flow attacks.

3.7.8 Debug Interfaces

The idea of using the debug interface (i.e., PTI in our work) for different purposes other than its intended usage has already been introduced and explored in several other studies, especially in the field of fault-tolerant computing [97, 98]. Fidalgo et al. [97] proposed a mechanism that can inject faults to the target system by accessing internal resources such as registers and memory via the interface. Another study presented an on-line fault detection technique that utilizes the interface to retrieve runtime information in a non-intrusive way [98]. Lately Lee et al. [13] discussed how the interface can be used to enhance the security of a system. They used the interface to transfer the internal cache data from the CPU into their external security device for memory traffic monitoring. Lastly, as mentioned in Section 3.6, the mechanisms to detect CRAs using the PTI outputs have also been proposed [99, 100]. These studies are conceptually similar to ours in that they exploit information flow that comes out of the debug interface without affecting the architecture of the CPU. However, none of these suggested the use of the interface to protect the OS kernel against the code injection attacks.

3.7.9 Architectural Supports for Shadow Stacks

As mentioned in Section 3.4, the idea of a hardware-based shadow stack has been explored [75, 101, 102, 103, 99]. While the design of the hardware-based shadow stack, which is also known as *Secure Return Address Stack (SRAS)*, is rather straightforward, handling of the `setjmp` and `longjmp` has been challenging. While Earlier work only proposed some mechanisms [103], SmashGuard [75] later addressed explicitly the challenge. Recently, Lee et al. [99] have also proposed a hardware shadow stack for an ARM-based system using the PTI traces like KS-Stack. However, all these mechanisms have been designed to detect the attacks on applications, so they did not need to overcome the challenges we stated before in Section 3.4. On the other hand, KS-Stack is designed to manage the contents of the shadow stacks correctly and securely without relying on the OS kernel, as it is designed to detect the attacks on OS kernels. It is worth noting that KS-Stack did not need to handle `setjmp` of `longjmp` as the OS kernels do not use them in general.

3.8 Summary

This chapter presented Kargos, the first mechanism to detect all kernel code injection attacks without mediating all accesses to the page tables. While existing solutions either detect only certain forms of the attacks or consume considerably the computing resources on the CPU, Kargos detects all the attacks with nearly zero performance cost. In our design, we provide mechanisms, called the atomic code blocks and the reporting protocol, for Kargos to protect the integrity of special registers in the CPU and to extract the values of them without relying on the service of the OS kernel. Unlike the hypervisor or any other software that runs on the CPU, our external hardware components technically have no means to access these registers directly. On top of these architectural supports, we also presented the design of KS-Stack, a means to defeat the ROP attacks on OS kernels. With this application, we could see that the new architectural features could be used for more than one purposes. For this reason, a system that includes Kargos would be able to adopt KS-Stack to detect the ROP attacks easily. In addition, we could expect more applications to detect the advanced control-flow attacks using the proposed architectural supports.

Chapter 4

Data-Flow Isolation

4.1 Motivations

Memory corruption vulnerabilities are the root cause of many modern attacks. To defeat such attacks, many security features have been commoditized, including NX-bit (No-eXecute), Supervisor Mode Execution Protection (SMEP), Supervisor Mode Access Prevention (SMAP), Memory Protection Extension (MPX), which have provided a strong foundation for security in today's computer systems. However, while these hardware-based security features are very efficient, they do not provide adequate protection against modern, complex memory-corruption-based attacks. For example, NX-bit can eliminate simple forms of code injection attacks, but cannot stop code-reuse attacks such as return-to-libc attack [104], return-oriented programming (ROP) [70], COOP [82], and non-control data attacks [105, 106, 107].

To defeat these new attacks, researchers continue to develop new hardware-based mechanisms. For example, hardware-based shadow stacks have been proposed to protect return addresses from tampering by adversaries [108, 103, 75]. Hardware-based control-flow integrity (CFI) has also been proposed to prevent code-reuse attacks, with various trade-offs [109, 110, 111, 112]. Furthermore, a number of other approaches have been proposed to eliminate the root cause of these memory corruption vulnerabilities [113, 114, 115, 116].

Our work also aims to prevent memory corruption based exploits. Towards this end, we take the direction of developing a new hardware feature that provides both flexibility (i.e., applicable to broad use cases) and performance (i.e., very efficient when activated and otherwise near-zero performance overhead on common execution paths). Our key observation is that even with hardware support, enforcing memory safety for the whole application is still too expensive for practical use, e.g., WatchDogLite [115] imposes 29% slowdown on SPEC CINT 2006 benchmarks. To further reduce performance overhead, one promising direction is to divide the memory into different regions—one for sensitive data (e.g., function pointers) and the other for the rest (e.g., application data). Then, we enforce memory safety only over the sensitive region [117, 118, 107]. There are two major advantages of this approach. First, sensitive data is usually a smaller set than normal data, and less data implies fewer checks and less performance overhead. Second, the safety of memory operations over sensitive data is easier for static verification. For example, because pushing/popping data onto/from stack is always safe, once we isolate the stack slots used to store return addresses, we can guarantee memory safety for return addresses without any runtime check. With these two advantages, we can significantly reduce the number of runtime checks, thereby making memory safety more affordable. However, implementing this strategy on commodity hardware is non-trivial due to the lack of *an efficient, fine-grained mechanism for data isolation*.

Table 4.1 compares existing software-based (top half) and hardware-based (bottom half) isolation mechanisms on *commodity hardware*. The most apparent problem is that the two most efficient hardware-based mechanisms—segment in x86 and access domain in ARM processors, are absent on 64-bit mode. As a result, security solutions in modern processors must make a trade-off between security and performance—solutions that opt for performance, e.g., by using randomization based protection, are usually subject to information disclosure or brute-force based attacks [124, 125], while solutions that opt for security, e.g., by leveraging context switch or masking, usually yield poorer

Table 4.1: Comparison between HDFI *and other isolation mechanisms, based on (1) whether data shadowing is required, (2) whether context switch is required for data access, (3) whether liveness tracking is required, (4) is available on 64-bit mode, (5) whether they can be used for self-protection, (6) is vulnerable to information leak, and (7) performance overhead. Self-protection means whether the mechanism can be used to prevent attacks from the same privilege-level. Performance overhead is measured by comparing one instrumented read/write operation against a normal memory read/write operation.* [1]*There is no public benchmark result for ADI, so this conclusion is purely based on their presentation [119].*

	Mechanism	**(1)**	**(2)**	**(3)**	**(4)**	**(5)**	**(6)**	**(7)**
Software based	Randomization [118, 117]	Y	N	N	Y	Y	Y	low
	Masking [118, 117, 72]	Y	N	N	Y	Y	N	moderate
	Access control list [120, 121]	N	N	Y	Y	Y	N	high
Hardware based	x86 memory segment [122, 117]	Y	N	N	N	Y	N	low
	ARM access domain [123]	Y	Y	N	N	Y	N	moderate
	Virtual address space [107]	Y	Y	Y	Y	Y	N	high
	Privilege level	Y	Y	Y	Y	N	N	moderate
	Virtualization [52]	Y	Y	Y	Y	N	N	high
	TrustZone [62]	Y	Y	Y	Y	N	N	very high
	ADI [119]	N	N	Y	Y	Y	N	low[1]
	HDFI	N	N	N	Y	Y	N	low

performance [118, 72, 107].

However, even if we managed to bring back the segment and access domain, these mechanisms are still inadequate. Specifically, because they are all coarse-grained, if we want to isolate data at a smaller granularity (e.g., function pointers) and preserve a program's original memory layout, then we must perform *data shadowing*. Unfortunately, data shadowing breaks data locality and requires extra steps to retrieve the shadow data. This introduces additional performance overhead [74]. Furthermore, data shadowing also introduces unavoidable memory overhead.

To overcome these limitations, we propose *hardware-assisted data-flow isolation* (HDFI), a new fine-grained data isolation mechanism. To eliminate data shadowing, HDFI enforces isolation at machine word granularity by virtually extending each memory unit with an additional tag. We choose to enforce isolation at this granularity because it balances the memory overhead (finer granularity requires more spaces for tags) and application requirements—a majority of sensitive data like pointers are at this granularity, and the rest can be easily aligned through software approaches. Please also note that HDFI's tags are associated with memory units' *physical addresses*, so attackers cannot tamper or bypass the protection by mapping the same physical page to different virtual addresses. Moreover, instead of using static partition, the tag is defined by data-flow. Inspired by the idea of data-flow integrity [15], HDFI defines the tag of a memory unit by the last instruction that writes to this memory location; then at memory read, it allows a program to check if the tag matches what is expected. This capability allows developers to enforce different security models. For example, to protect the *integrity* of sensitive data, we can enforce the Biba Integrity Model [126]. In particular, we can use the tag to indicate integrity level (IL) of the corresponding data: sensitive data has `IL1` and normal data has `IL0`. Next, we assign IL to write operations based on the data-flow. That is, we use static analysis to identify write operations that can manipulate sensitive data, and allow them to set the memory tag to `IL1`; all other write operations will assign to the tag to `IL0`. Finally, when loading sensitive data from memory, we check if the tag is `IL1` (see section 4.3 for a concrete example). HDFI can also be used to enforce *confidentiality*, i.e., the Bell–LaPadula Model [127]. For instance, to protect sensitive data like encryption keys, we can set their tag to `SL1` (secret level 1), and enforce that all untrusted read operations (e.g., when copy data to an output buffer) can only read data with tag `SL0`.

We implemented a prototype of HDFI by extending the RISC-V instruction set architecture (ISA) [128], an open-source, license-free ISA that is designed with direct hardware implementation and practical applications in mind. Our prototype implementation was designed to support one-bit tag for two reasons: (1) it limits the amount of required resources; and (2) as discussed above, in most security applications, one-bit tags are sufficient.

To evaluate the performance of HDFI, we instantiated it on the Xilinx Zynq ZC706 evaluation board [129] and ran several benchmarks including the SPEC CINT 2000 [130] benchmark suite. Evaluation results show that the performance overhead caused by our modification to the hardware is low ($< 2\%$).

To summarize, this chapter makes the following contributions:

- **Design**: We present a new hardware security mechanism, which is general, efficient, backwards compatible, and only requires small hardware modification. We also present optimization techniques to minimize HDFI's performance impact on normal operations (section 4.4).
- **Implementation**: We implemented the proposed hardware with realization on FGPA board. (section 4.5).
- **Evaluation**: We quantitatively evaluated the performance impact of HDFI and the effectiveness of our optimization techniques. (section 4.6).

4.2 Threat Model and Assumptions

In this work, we focus on preventing memory corruption based attacks; therefore we follow the typical threat model of most related work. That is, we assume that software may contain one or more memory vulnerabilities that, once triggered would allow attackers to perform arbitrary memory reads and writes. We do not limit what attackers would do with this capability, as there are many different attack vectors given this capability. As a hardware-based solution, we also do not limit where the vulnerabilities are: they can be in user-mode applications, OS kernel, hypervisor, etc. However, we assume all hardware components are trusted and bug free, so attacks that exploit hardware vulnerabilities, such as the row hammer attack [131], are out-of-scope.

Similar to NX-bit, HDFI requires software modifications to obtain its benefits. This can be done in many ways: manual modification, compiler-based modification, static binary rewriting, dynamic binary rewriting, etc.

4.3 Background and Related Work

This section provides the background of HDFI and compares HDFI with related work.

4.3.1 Data-flow Integrity

The goal of HDFI is to prevent attackers from exploiting memory corruption vulnerabilities to tamper/leak sensitive data. To achieve this goal, we leverage data-flow integrity (DFI) [15]. DFI ensures that the runtime data-flow cannot deviate from the data-flow graph generated from static analysis. In particular, DFI assigns an identifier to each write instruction and records the ID of the last instruction that writes to a memory position; then at each read instruction, DFI checks whether the ID of the last writer belongs to the set allowed by static analysis. Take Example 4.1. This code snippet contains a buffer overflow vulnerability at line 6, which allows attackers to use `strcpy()` to overwrite the return address saved at line 3 and launch control-flow hijacking attacks. Such attacks can be prevented by checking if the return address read at line 8 is defined by the store instruction at line 3.

In HDFI, we extend the ISA to perform DFI-style checks with hardware. Specifically, we leverage memory tagging to record the last writer of a memory word and provide new instructions to set and check the tag. However, instead of trying to fully replicate DFI, which would require supporting arbitrary tag size, we focus on providing isolation,

```
main:
  add      sp,sp,-32
 ⋆sdset1   ra,24(sp)
  ld       a1,8(a1)          ; argv[1]
  mv       a0,sp             ; char buff[16]
  call     strcpy            ; strcpy(buff, argv[1])
  li       a0,0
 ⋆ldchk1   ra,24(sp)
  add      sp,sp,32
  jr       ra                ; return
```

Example 4.1: *A typical stack buffer overflow example, in RISC-V assembly, in which* HDFI *prevented by replacing load and store instructions with two new load and store instructions (line 3 and 8).* `strcpy()` *at line 6 can overflow the return address saved at line 3, and* HDFI *can accordingly detect the overflow when it is loaded back at line 8.*

i.e., using a one-bit tag to indicate the trustworthiness of the writer. Using the same example, HDFI can be utilized to prevent the attack by (1) using a new instruction `sdset1` (store and set tag) to set the tag of memory used to store return address to `1` (line 3); and (2) when loading the return address from memory for function return, using another instruction `ldchk1` (load and check tag) to check if the memory tag is still `1`. Since normal store instructions (e.g., `sd`) would set the tag to `0`, if attackers try to overwrite the return address, the `ldchk1` instruction would fail and generate a memory exception.

4.3.2 Tag-based Memory Protection

Tag-based memory protection is not new and has been explored in many previous works. For example, lowRISC [132] uses a 2-bit tag to specify if a memory address is readable and writable. Loki [133] also allows developers to specify permission with a memory address, but is more flexible, as the permission is related to the current protection domain. The problem with these approaches (including the Mondriaan protection model [134]) is that, although the objects (memory addresses) are fine-grained, the subjects are still coarse-grained—the access permissions are applied to the whole program or the whole protection domain. However, the subjects are individual instructions in HDFI.

An alternative approach is to associate the access permission with pointers instead of memory locations. For example, Watchdog [114] and the application data integrity (ADI) [119] mechanism on SPARC M7 processors allow a program to associate memory addresses and pointers with versions (tags) and require that when accessing the memory the version of the pointer must match the version of the memory. The tricky part of this approach is how to maintain the tag of a pointer, because every pointer should have two tags: one indicating the tag of the target memory, and the other indicating the tag of the memory where the pointer is stored. Without this, attackers can still tamper with the pointers. Watchdog handles this by using shadow memory to maintain the first type of tags, but it is unclear whether or how ADI handles this issue.

Write integrity test [135] is another tag-based memory safety enforcement mechanism. It enforces that each write operation (instead of pointer) can only write to objects that are allowed by the static data-flow graph. However, since the integrity test is only enforced on write operations, WIT can only enforce data integrity, but not data confidentiality.

A common issue with all the aforementioned approaches is that they must track the liveness of memory objects, which makes the protection more complicated. For instance, in Example 4.1, to protect the return address, all aforementioned systems must tag the memory used for return address at prologue. Here we must pay special attention to the order of tagging and store: if store happens before tagging, the system would be vulnerable to time-of-check-to-time-of-use (TOCTTOU) attack, because the address might be modified unless the two operations are guaranteed to be atomic. Then, after the function finishes execution and returns, the current stack frame is *freed*, so the old memory position used to store the return address must be unprotected for future re-use. Here is another tricky part—if the capability system is location-based, or does not assign a new version for every memory allocation (which is very challenging for fixed tag size), then it would be subject to use-after-free (UAF) based attacks. Moreover, for software that heavily utilizes custom memory allocators, such as browsers and OS kernels, tracking object allocation is non-trivial. Fortunately, HDFI does not need to track liveness of memory objects.

Among existing hardware features, Minos [109] and CHERI [116] are the closest to HDFI. Specifically, Minos uses one-bit tags to indicate the integrity of code pointers and updates the tag based on the Biba model [126]. CHERI [116] also uses one-bit tags to indicate whether a memory address stores a valid capability (fat pointer). This bit can only be set when the memory content is written by a capability-related instructions and is cleared when written by normal store instructions. Comparing to them, the advantage of HDFI is flexibility—besides pointers, HDFI can also be used to protect generic data like `uid`; and along with the Biba model, HDFI can also be used to enforce the Bell–LaPadula model [127].

4.3.3 Tag-based Hardware

Memory tagging is widely used for dynamic information flow tracking (DIFT), which can be very expensive when purely done in software [136]. For this reason, numerous hardware solutions have been proposed, including pure DIFT-oriented [137, 138, 109, 139], and more general, programmable metadata processing [140, 141, 142, 143]. The most significant difference of HDFI from these solutions is our emphasis on minimizing hardware changes so as to make HDFI more likely to be adopted in practice. In particular, HDFI does not require modifying register files, ALU, main memory, or the bandwidth between cache and main memory. More importantly, instead of requiring half of all physical memory dedicated to store tags (i.e., an overhead of 100%), HDFI only impose 1.56% memory overhead.

4.3.4 Memory Safety

Since memory safety issues are the root causes of many attacks [1], researchers have proposed many solutions to address this problem, including automated code transformation [144], instrumentation-based [15, 135, 16, 145], and hardware-based [113, 114, 115, 146, 116]. The biggest hurdle for adopting these solutions is their performance overhead—even with hardware assistance, the average overhead is still 29% on benchmark workloads [115]. To help further reduce the overhead, HDFI is designed to enable another optimization direction—using isolation to limit the protection scope and only enforcing memory safety over the isolated data. Such data could be security sensitive, e.g., code pointers [117], generic pointers [147, 116], or important kernel data [107]. It could also be data that can be statically proved to be memory safe, e.g., safe stack [117]. We believe such a combination would allow us to build powerful yet efficient solutions to eliminate all memory corruption based attacks.

4.4 HDFI Architecture

In this section, we present the design of HDFI, which includes two major components: the ISA extension and the memory tagger. Our current design tags memory at machine-word granularity because most sensitive data we want to protect are of this size (e.g., pointers). For data not of this size, we can manually extend the size, or leverage compilers. To prevent attackers from creating inconsistent views of data and its corresponding tag and launching TOCTTOU attacks in a multi-core/-processor system, we require all HDFI instructions to be *atomic* (i.e., data and tag must always be loaded and stored together) and comply with the same cache consistency model as other memory accessing instructions. To avoid changing the main memory system and the data link between main memory and the processor, our current design stores all the tag information at a dedicated area called *tag table*. In our current design, tag table is allocated and initialized by the OS kernel during boot, similar to how Intel

SGX reserves the secure pages (i.e., EPC pages) for enclaves [146]. Once allocated, the memory region for the tag table will be protected from malicious modification (subsection 4.4.4).

4.4.1 ISA Extension

To enforce DFI, the authors added two high-level instructions: `SETDEF` and `CHECKDEF` [15]. Since HDFI only supports one-bit tags, in order to allow programs to use DFI-style checks to enforce the integrity/confidentiality level of memory contents, we introduce three new instructions:

- `sdset1 rs,imm(rb)`: store word and set tag to `1`.
- `ldchk0 rd,imm(rb)`: load word and check if tag equals `0`.
- `ldchk1 rd,imm(rb)`: load word and check if tag equals `1`.

Note that we do not have an instruction that explicitly sets the tag to `0`. Instead, HDFI implicitly sets the tag of the destination memory to `0` when written by regular store instructions. However, HDFI preserves the semantics of regular load instructions, i.e., tag is not checked on regular load operations. To check the tag bit of the target memory region, HDFI provides `ldchk0` and `ldchk1`. To enable the OS kernel to capture tag mismatch, we also introduced a new memory exception, which is similar to other memory faults except for the error code.

HDFI also provides a special instruction alias `mvwtag` [1] for copying the memory from a source to a destination along with the corresponding tag bits. This special operation is necessary to achieve optimal performance in modern system software. Specifically, modern OS kernels like Linux use copy-on-write (CoW) to share memory between the parent process and its child processes. However, if we use normal `sd`

[1]Since we do not extend general register files with tag, this operation is an alias for two instructions: load data and tag from source into a special register then store them to the destination.

operations to perform the copy, it could break HDFI-protected applications because the tag information is lost; on the other hand, we also cannot use `sdset1` because it allows attackers to abuse this feature to tag arbitrary data. To solve this problem, we introduced the `mvwtag` instruction to allow OS kernels to copy data while preserving the tag. Please also note that because `memcpy` can cause memory corruption, we do not recommend using `mvwtag` to implement `memcpy` unless the developer can guarantee memory safety for all the invocations of `memcpy`.

4.4.2 Memory Tagger

Our hardware extension is similar to lowRISC [132]. Specifically, to simplify the implementation of the new instructions and support atomicity in a multi-core/-processor system, we modified the interface between the processor core and the cache system (including the coherence interconnect) to associate each data with its tag. In particular, when the processor core executes a memory related instruction such as `sd`, `sdset1` or `ld`, it sends a request to the data cache(s). This request includes a data field and a command field. HDFI adds one tag bit to the data field, so for every memory write request, data is always stored with the tag; and for memory load requests, tags can be (not always, see subsection 4.4.3 for detail) loaded with data.

To facilitate this, we augmented the caches to hold the tags for the cached memory units, as shown in Figure 4.1. To hold the tag bits for the cached memory units, the caches have a one-bit register for each machine word to store the corresponding tag. When the processor core sends a store request, the L1 cache can simply update the data and tag value with the incoming value from the core; and when the core sends a read request, the L1 data cache provides the core with the tag bit, with which the core can check whether the tag matches expected value or not.

While the L2 cache can also be augmented similarly to hold the tags for each memory unit, we believe it is not feasible to add the tag bits physically to the external main memory. For this reason, we added an additional module DFITAGGER in between

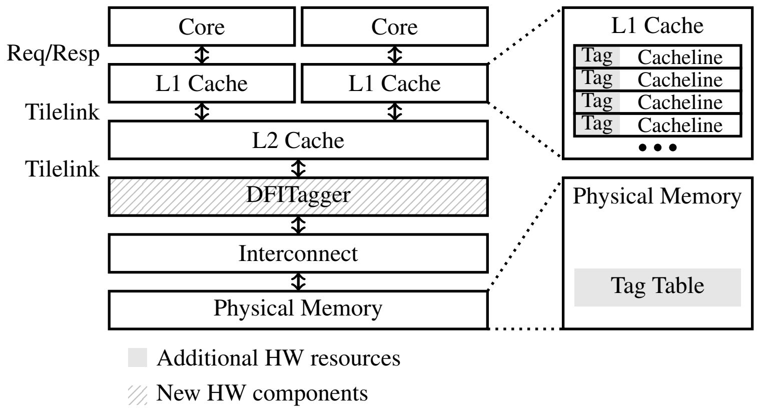

Figure 4.1: Design of HDFI. *The processor core and caches are augmented and the* DFITAGGER *is added.*

the L2 cache and the main memory, which decomposes memory accesses from the L2 cache to separate data accesses and tag accesses. Data accesses are handled as usual and tag accesses are handled as follows. HDFI preserves a memory chunk to be used as tag table (Figure 4.1), which acts as a huge bit vector to store tag bits. When the L2 cache issues a memory access, DFITAGGER maps the physical address to a table entry of the tag table and generates a tag access.

4.4.3 Optimizations

Unfortunately, the additional memory accesses to the tag table introduce non-negligible performance overhead. More specifically, without any optimization, HDFI will double the memory accesses because for every cache miss, DFITAGGER needs to issue one data access and another tag access. To minimize this impact, we developed several optimization techniques.

4.4.3.1 Tag Cache

The most straightforward way of reducing the overhead is caching, so we introduced a tag cache within the DFITAGGER to exploit the locality of memory accesses. Moreover, tag cache also allows DFITAGGER to fetch a set of tags from the main memory in the cache line granularity to reuse the existing memory interface. For example, a cache line in the Rocket Core is 64 bytes. To handle one cache miss, DFITAGGER only needs 8 tag bits (one bit per eight bytes), but for the fixed size of memory interface, it has to fetch 64 bytes from the tag table. In fact, this 64-byte unit, which we call one *tag table entry*, naturally stores the tags for a 4 KB memory block; so tag cache allows us to generate only one memory access per 4 KB data access.

4.4.3.2 Tag Valid Bits

The second optimization technique takes advantage of the fact that most of the memory loads are *not* checked, so there is no need to always refill the cache line with corresponding tag bits. Leveraging this observation, we add a *Tag Valid Bit* (TVB) to each memory unit in the caches to further reduce unnecessary accesses to the tag table. TVB is updated as follows. When the cache has to refill a line but the request from the inner cache or the processor core does not explicitly asks for tag bits, the cache generates a refill request to the outer cache or DFITAGGER, and clears the TVB for the memory units in the line. Later, if an incoming load (with tag) request hits in the cache, but the TVB for the corresponding memory unit is not set, the cache will refill the line again with the valid tags. Note that any write hit will set the TVB because store operations always update the tag bit. Finally, when a cache line is evicted and written back to main memory, the cache forwards TVB to DFITAGGER, so the later can update the tag cache accordingly.

4.4.3.3 Meta Tag Table

The third technique leverages the fact that most of the memory units are tagged with `0` and only a few ones will be tagged with `1`. This means that most tag table entries would be filled with `0`. To take advantage of this observation, DFITAGGER maintains a *Meta Tag Table* (MTT) in the main memory and a *Meta Tag Directory* (MTD) as a register.

Each bit of the MTT entries is set to `1` if the corresponding tag table entry contains `1`, and each bit of MTD is set to `1` if the corresponding MTT entry has `1`. Utilizing them, DFITAGGER can avoid fetching tag table entries from the main memory if they are filled with `0`. It also enables DFITAGGER to avoid (1) updating the tag table entry for a given write miss if that entry is filled with `0`; and (2) write back to main memory if both the evicted tag cache and the main memory copy are filled with `0`.

4.4.4 Protecting the Tag Tables

The design of HDFI requires that the tag table and the meta tag table in the main memory are protected from the malicious modifications. To do so, we leverage the fact that DFITAGGER is sitting between the cache and the main memory, hence we can use it to mediate all modifications to the main memory. That is, once the memory chunk used for tag tables are assigned to DFITAGGER, it drops any access to this memory chunk. Because tag is always provided by DFITAGGER, this effectively prevents any malicious modifications to the tag tables. Note that our current design cannot prevent DMA-based attacks; we will discuss this issue in section 4.7.

4.5 Implementation

Table 4.2: Components of HDFI *and their complexities in terms of their lines of code.*

Components	Language	Lines of Code		
		Modified	Added	Total
Architecture	Scala (Chisel)	395	1,803	2,198
Assembler	C	-	16	16
Linux Kernel	C	8	52	60
Total		403	1,871	2,274

In this section, we provide the implementation detail of HDFI. Table 4.2 shows the lines of code used to implement HDFI, excluding empty lines and comments.

4.5.1 Hardware

We implemented a prototype of HDFI by modifying the Rocket Chip Generator [148]. The generated system includes a Rocket Core [149] as its main processor, which has 16KB of L1 instruction and data caches. Modifying the generator itself instead of a generated instance allows us to generate and evaluate multiple versions of HDFI with various features and parameters, e.g., different optimization techniques and configuration parameters.

4.5.1.1 ISA Extensions

Following the design pattern of RISC-V, we assign a new opcode to our new instructions that is similar to the RV64I load/store instructions [128].

`sdset1`: We extend the memory request unit's data field by one-bit to include the tag. To determine whether the tag should be `0` or `1`, we introduce a new configuration to the set of control signals for memory command type that is unique to `sdset1`.

`ldchkx`: We add a new, one-bit field to the memory response unit for the tag bit loaded with the machine word. To determine whether the tag bit should be loaded, we assigned a new memory command to these two instructions. Upon a valid response from cache, HDFI compares the tag to the expected value. This expected value is extracted from bit `12` of the `ldchkx` instruction. A tag mismatch generates a new memory exception; otherwise, the pipeline continues normally.

`mvwtag`: At the execution stage, HDFI first calculates the source address from the second register's value and the immediate offset using the ALU, and sends out a memory read request to load the data and tag. The result is stored in a new internal register that is capable of storing both data and tag. Simultaneously, HDFI calculates the destination address from the destination register's value and the same offset using a separate adder. Finally, we issue a memory store request to store the internal register's

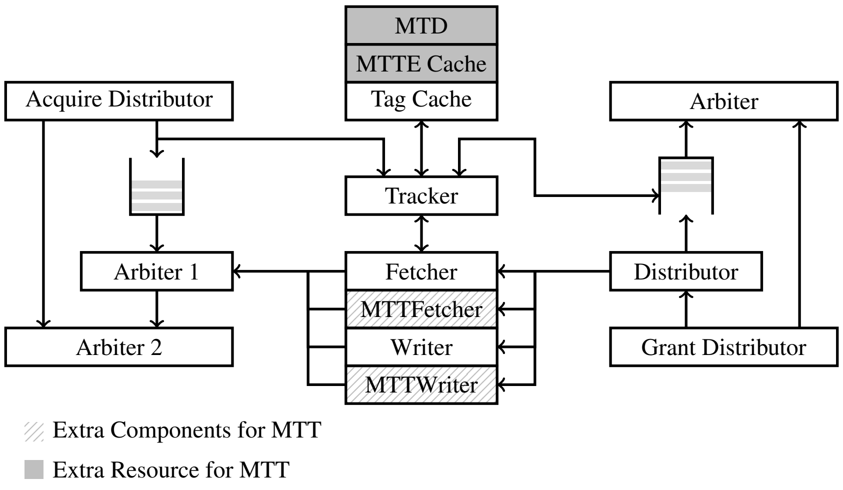

Figure 4.2: A simplified diagram of DFITAGGER *on a Rocket Chip.*

data and tag to the destination address.

4.5.1.2 DFITAGGER

To avoid adding the tag bits physically to the main memory, which is usually a set of DRAMs, we implemented DFITAGGER to translate memory accesses with tags from inner caches into data accesses and tag accesses. Figure 4.2 shows the DFITAGGER we implemented for the Rocket Chip. The DFITAGGER is designed to handle the memory accesses that comply with the *TileLink* protocol which the rocket chip uses to implement the cache coherence interconnect. Among the five channels that the protocol defines, DFITAGGER handles two of them because they are used to connect the L2 caches and the outer memory system.

To initiate a memory access, the inner cache generates one or more *beats* of transaction through the *Acquire* channel, and the DFITAGGER selectively intercepts the beats using the *Acquire Distributor*. When the option `tagger` is enabled, the Acquire Distributor bypasses the device accesses, drops the access to the tag table or meta tag table (for protection purpose), and forwards all the transactions heading to the memory to the *Acquire Queue*, which simply forwards the incoming transactions to memory.

The *Acquire Arbiter1* drops all the tag bits in the transactions; the resulting memory accesses only contain the data part of the incoming accesses.

In the mean time, the *Tracker* duplicates the required field of incoming transactions, including the tag bits, the transaction id, the type of the transaction, and the address. When the incoming transaction writes to memory, the Tracker updates the corresponding tag bits in the tag table with the tag bits in the transaction. To do this, the Tracker first check the *Tag Cache*, and uses the *Fetcher* and the *Writer* modules to fetch and evict tag table entries.

Handling memory read accesses is similar, but the Tracker need to intervene in the *Grant* channel as well. In the Rocket Chip, the memory interface (which is a protocol converter) uses the Grant channel to provide the caches with the read data. To attach the tag bits to the Grant transactions, the Acquire Queue changes the transaction id of read accesses so that the corresponding Grant transactions are forwarded to the *Grant Queue*. In the mean time, the Tracker accesses the Tag Cache and uses other modules to prepare the corresponding tag bits. Once the tag bits become available, the Grant Queue forwards the transaction from the memory interface, after changing the transaction id back to the original one and attaching the correct tag bits for each machine word.

4.5.1.3 Tag Valid Bits

To reduce the number of tag table accesses, HDFI adds a TVB for each machine word in the caches. Using TVB, the cache can avoid fetching the tag bits when it refills a cache line. To take advantage of this, the cache uses the `union` field of an Acquire transaction to mark if the response to the transaction should have valid tag bits or not. The Acquire Distributor then uses this field to decide whether a transaction could be directly forwarded to the *Acquire Arbiter2* and bypass the Acquire Queue.

The location of TVBs is also important. A simple solution is to put the TVBs in the metadata array, where the cache holds the *cache tags* and the coherence information. However, this approach would increase the latency of write hits because the cache has to update the metadata for every write operation. To address this issue, we choose to put a *tag fetched bit* in the metadata array for each cache line and extend the size of the data array to store the TVBs for each word. The tag fetched bit is set/cleared by the miss handler, which is called *MSHR* in the Rocket Chip. When the handler fetches the cache line with tags, it sets the bit; otherwise the bit is cleared. Since every write operation should update the tag, the cache also sets the TVB whenever a machine word is written.

Adding TVBs also requires the DFITAGGER to consider a memory write access whose tag bits are partially valid. To handle this, the cache attaches the TVBs for each machine word to the Acquire transactions for memory writes. With the TVBs, the DFITAGGER can selectively update the tag bits in the corresponding tag table entry.

An important drawback of this implementation is that the cache refills a cache line to handle an incoming load with tag access even when the TVB of the requested machine word is set, but if the Tag Fetched Bit is not set. We believe that we can avoid these cache refills by augmenting the miss handler, by letting it to consider the TVBs before evicting and refilling the cache, but the current implementation does not include such feature.

4.5.1.4 Meta Tag Table

Enabling the Meta Tag Table adds the shaded components and resource in Figure 4.2 to the DFITAGGER. When handling an incoming tag table read access, the Tracker checks whether the MTT cache and the tag cache has a matching entry. If the Tracker fails to find a matching tag table entry, it checks the MTD and the matching MTT entry (loaded into MTT cache if does not exist) to see if the corresponding tag table entry is all zero. If so, the Tracker handle the incoming tag table access without really fetching the entry from the memory. To minimize the miss penalty, the *MTTFetcher* and the *MTTWriter* handles the access to the MTT in the memory in parallel with the existing Writer and Fetcher.

After updating the tag table entry and the MTT entry, the Tracker checks if it can clear the corresponding MTT entry bit and MTD bit. In particular, the Tracker clears the corresponding bit in MTT entry if the updated tag table entry is filled with zeros, and clears the MTD bit if the MTT entry is filled with zeros.

4.5.2 Software Support

To utilize HDFI, we made the following changes to the software.

4.5.2.1 Assembler

We modified the GNU assembler (`gas`) so that it recognizes the new instruction extension and can generate the correct binary.

4.5.2.2 Kernel Support

Our modifications to the OS kernel include three parts. First, we modified its exception handler to recognize the new tag mismatch exception. To handle this exception, we reused the same logic as normal load/store faults, i.e., generate a segment fault (`SIGSEGV`) for user mode applications, and panic if the exception happens in kernel space. Second, as mentioned in section 4.4, we implemented a special memory copy routine with the new `mvwtag` instruction and modified the CoW handler to invoke this routine to copy page content, so that the tag information are preserved. Last, we added routines to allocate the tag table and meta tag table, and initialize the DFITAGGER with the base addresses of the tables.

4.6 Evaluation

In this section, we evaluate our prototype of HDFI by answering the following questions:

- **Correctness**. Does our prototype comply with the RISC-V standard (i.e., no backward compatibility issue)? (subsection 4.6.1)

- **Efficiency**. How much performance overhead does HDFI introduce compared to the unmodified hardware? (subsection 4.6.2)

Experimental setup. All evaluations were done on the Xilinx Zynq ZC706 evaluation board [129]. The OS kernel is Linux 3.14.41 with support for the RISC-V architecture [150]. Unless otherwise stated, all programs (including the kernel) were compiled with GCC 5.2.0 (`-O2`) and binutils 2.25, with a set of patches to support RISC-V (commit `572033b`) and default kernel configuration of RISC-V. While the board is equipped with 1GB of memory, the Rocket Chip can only use 512MB because the co-equipped ARM system requires 256MB. At boot time, the kernel reserves 8MB for tag tables and 128KB for the meta tag table, respectively. Following the environment that the RISC-V community built, we use the *Frontend Server* that runs on the ARM system and the *Berkeley Boot Loader* that runs on the Rocket Chip to boot vmlinux. The Rocket Chip accesses an ext2 file system in an SD card via the Front-end Server.

Although the tape-out Rocket Core chip can operate on 1GHz or higher, the synthesized FPGA on the ZC706 board can only operate at the maximum frequency of 50MHz. In addition, because the L2 cache is not mature enough for memory-mapped IO [151], we only evaluated with the L1 caches. In place of the L2 cache, we used the *L2BroadcastHub* that interconnects the L1 caches and the outer memory system. Due to the above limitation and the memory limitation of the evaluation board, we were not able to run most SPEC CINT 2006 benchmarks, so we used the much lighter SPEC CINT 2000 [130]. For SPEC CINT 2000, some benchmarks (`gzip` and `bzip`) cannot run successfully with the reference inputs. For these benchmarks, we adjusted the parameters of the reference inputs to reduce the size of the buffer they use to 3MB. We have annotated the results to clarify this.

We used pseudo-LRU (Least Recently Used) as the replacement policy for both tag and meta tag caches, and set the size of each cache to 1KB, allowing up to 16 entries of 512-bit cachelines.

4.6.1 Verification

HDFI passes the RISC-V verification suite provided by the RISC-V teams, which means our modifications to the RISC-V complies with the RISC-V standard so unmodified programs can still run correctly on our modified hardware.

4.6.2 Performance Overhead

In this subsection, we evaluate the performance impact of our hardware extension, as well as the effectiveness of our optimization techniques. This evaluation includes two part: the impact of new instructions on the processor core and the impact on memory access. Since HDFI did not introduce many changes to the pipeline of the processor core, the focus will be on memory access.

4.6.2.1 Pipeline

The `sdset1` and two `ldchk` instructions are treated identically to their normal store and load counterparts in the pipeline, with the exception of `ldchk` doing a comparison at the end of the memory stage. These three instructions can stall the pipeline in the same manner as their counterparts. However, the special register dedicated to `mvwtag` for preserving tags introduces a structural hazard to the pipeline. Because there is only one special register available, a series of `mvwtag` instructions have to wait for the previous `mvwtag` to finish, stalling the pipeline. Other memory instructions do not have to wait on previous ones to issue memory requests.

4.6.2.2 Memory Access

While the ISA extension does not affect the performance of the processor core, HDFI inevitably introduces additional memory accesses to fetch/update the tag table.

Micro benchmark. To measure the performance impact of these additional memory accesses and the logics to deal with them, we used `lat_mem_rd` from LMBench [78]

Table 4.3: Impact on memory bandwidth and read latency, with different optimization techniques. The load does not include tag check and store does not include tag set.

Benchmark	**Baseline**	**Tagger**	**TVB**	**MTT**	**TVB+MTT**
L1 hit	40ns	40ns (0%)	40ns (0%)	40ns (0%)	40ns (0%)
L1 miss	760ns	870ns (14.47%)	800ns (5.26%)	870ns (14.47%)	800ns (5.26%)
Copy	1081MB/s	939MB/s (13.14%)	1033MB/s (4.44%)	953MB/s (11.84%)	1035MB/s (4.26%)
Scale	857MB/s	766MB/s (10.62%)	816MB/s (4.79%)	776MB/s (9.45%)	817MB/s (4.67%)
Add	1671MB/s	1598MB/s (4.37%)	1650MB/s (1.26%)	1602MB/s (4.13%)	1651MB/s (1.2%)
Triad	818MB/s	739MB/s (9.66%)	802MB/s (1.96%)	764MB/s (8.8%)	803MB/s (1.83%)

to measure memory access latency and STREAMBench [42] to measure memory bandwidth. Table 4.3 shows the result of the five configurations. The first row shows that HDFI does not affect the cache access latency. As the system operates at 50MHz, the 40ns latency means that it takes two clock cycles to read from the L1 cache. The second column shows that HDFI does increase the memory access latency. When TVB is enabled, DFITAGGER simply bypasses the incoming memory read access unless it explicitly requests the tag bits. However, the access should be examined by the Acquire Distributor and the Grant Distributor (Figure 4.2), which adds 2 clock cycles latency. For memory bandwidth, our results also show that the optimizations we implemented can effectively reduce overhead.

SPEC CINT 2000. In addition to the micro benchmarks, we also ran a subset of SPEC CINT 2000 benchmarks on the five configurations of HDFI, *without* any security applications (i.e., no load check and no `sdset1`). Table 4.4 shows that even though the unoptimized version of HDFI causes non-negligible performance overhead, our optimizations successfully eliminated a large portion of overhead. Specifically, since there is no load check, TVB eliminated all read access requests to the tag table; and since there is no `sdset1`, MTT eliminated all the write access to the tag table. Table 4.5 shows the number of memory accesses reduced by TVB and MTT. Please note that the 0.11% performance gain on `mcf` is due to fluctuations.

Table 4.4: Performance overhead of a subset of SPEC CINT 2000 benchmarks. Due to the limited computing power of the Rocket Chip on FPGA, we chose relatively lighter benchmark. In addition, to be fair, we included relatively memory bound benchmarks. According to a paper [152], 181.mcf, 175.vpr and 300.twolf are memory bound and showing higher overhead. We used reduced version of reference input to run 164.gzip and 256.bzip2.

Benchmark	**Baseline**	**Tagger**	**TVB**	**MTT**	**TVB+MTT**
164.gzip	963s	1118s (16.09%)	984s (2.18%)	1029s (6.85%)	981s (1.87%)
175.vpr	14404s	18649s (29.51%)	14869s (3.26%)	15513s (7.71%)	14610s (1.43%)
181.mcf	8397s	11495s (36.89%)	8656s (3.08%)	9544s (13.66%)	8388s (−0.11%)
197.parser	21537s	25005s (16.11%)	22025s (2.27%)	23177s (7.61%)	21866s (1.53%)
254.gap	4224s	4739s (12.19%)	4268s (1.04%)	4500s (6.53%)	4254s (0.71%)
256.bzip2	716s	820s (14.52%)	735s (2.65%)	742s (3.63%)	722s (0.84%)
300.twolf	22240s	28177s (26.71%)	22896s (2.97%)	23883s (7.37%)	22323s (0.36%)

Table 4.5: The number of total memory read/write access from both the processor and DFITAGGER.

Benchmark	**Type**	**Baseline**	**Tagger**	**TVB**	**MTT**	**TVB+MTT**
164.gzip	**Read**	590M	799M (35.25%)	606M (2.71%)	589M (−0.17%)	588M (−0.34%)
	Write	380M	1,217M (220.26%)	453M (19.21%)	1,017M (167.63%)	378M (−0.53%)
175.vpr	**Read**	9,816M	17,200M (75.15%)	10,930M (11.35%)	9,760M (−0.57%)	9,792M (−0.25%)
	Write	7,908M	37,480M (373.83%)	12,420M (57.06%)	31,890M (303.16%)	7905M (0%)
181.mcf	**Read**	9,778M	14,310M (46.35%)	10,503M (7.41%)	9,778M (0%)	9,778M (0%)
	Write	5,588M	23,720M (324.33%)	8,490M (1.11%)	20,300M (263.15%)	5,588M (0%)
197.parser	**Read**	12,770M	17,610M (37.9%)	13,220M (3.52%)	12,850M (0.63%)	12777M (0.01%)
	Write	8,290M	27,490M (231.6%)	9,640M (16.28%)	24,440M (194.81%)	8299M (0.11%)
254.gap	**Read**	2,233M	2,872M (28.61%)	2,239M (0.27%)	2,225M (0%)	2,206M (−1.21%)
	Write	1,594M	4,237M (165.81%)	1,701M (6.71%)	3,926M (146.3%)	1,592M (−0.13%)
256.bzip2	**Read**	228M	390M (71.05%)	268M (17.54%)	229M (0.44%)	229M (0.44%)
	Write	249M	896M (259.84%)	407M (63.45%)	730M (193.17%)	249M (0%)
300.twolf	**Read**	13,600M	22,350M (64.34%)	15,820M (16.32%)	13,600M (0%)	13,610M (0%)
	Write	13,680M	48,650M (255.63%)	22,510M (64.55%)	38,090M (178.43%)	13,610M (−0.51%)

4.7 Limitations and Future Work

Direct Memory Access (DMA). Since our current prototype of HDFI only handles memory accesses from the processor core, it is vulnerable to DMA-based attacks. Attackers can leverage DMA to (1) corrupt the data without changing the tag and (2) directly attack the tag table. To mitigate this threat, we could leverage features like IOMMU to confine the memory that can be accessed through DMA [52]. Alternatively, we can choose to add our own hardware module in between the interconnect and the memory controller such that all memory accesses would pass through the hardware module. By doing so, our hardware module would be able to determine whether or not the access is from DFITAGGER, thus prevents malicious access to the tag table. It is worth noting that similar hardware modules have already been introduced [9] and deployed in commodity hardware [146, 153].

Configurable Tag Table. Our current implementation completely blocks accesses to the tag table. Although this provides a stronger security guarantee, it also comes with some drawbacks. The first problem is that we cannot save the page to disk because the tag information will lost. To support these features, we must allow the kernel to access the tag table. However, to protect the tag table from tampering, we must implement some protection techniques like [64] or integrity measurements like [146]. Another drawback of our current design is that we must allocate the whole tag table in advance. In the future, we could provide other options for the OS kernel or the hypervisor to manage the tag table depending on the security requirement by users. On such a model, we can implement an on-demand allocation mechanism to reduce the memory overheads, i.e., we allocate the tag memory only when DFITAGGER modifies a tag entry.

Opportunities for Further Optimizations. Although the Rocket Chip Generator is a great tool for prototype verification, the Rocket Core is a very limited processor compared to x86 processors. With a more powerful processor core like the Berkeley out-of-order machine (BOOM) [154] and a more sophisticated cache, we could further reduce the memory access overhead using the following techniques.

Tag prefetch: Just like prefetching data that is likely to be used in the future due to program locality, we could also prefetch the tag. We could both prefetch the tag from DFITAGGER to avoid possible read miss hit due to TVB and prefetch the tag entries from the main memory when the bus is free.

Delayed check: Just like speculating a branch, as most tag checks should not triggering the exception, with an out-of-order machine we could speculate the execution even when the tag is not ready (i.e., TVB miss hit). By doing so, we could avoid stalling the pipeline and further reduce the overhead of HDFI.

Better cache design: In our prototype implementation, we did not extend our modification to the L2 cache. At the same time, as mentioned in subsection 4.5.1, out current design of TVB is not ideal, which may cause some obvious performance overhead for unoptimized programs. For future work, we plan to extend our modification to the L2 cache with better TVB implementation.

Dynamic Code Generation. Dynamic code generation is an important technique that has been widely utilized in browsers and OS kernels to improve performance. However, because this technique requires memory to be both writable and executable, it may be vulnerable to code injection attacks [155]; and unlike static code, it is not always possible to detect malicious modification to the generated code. In the future, we can perform tag checking for instruction fetching, i.e., provide a configuration flag that once enabled, only allows tagged memory to be fetched as code.

4.8 Summary

In this chapter, we have presented HDFI, a new fine-grained data isolation mechanism. HDFI uses new machine instructions and hardware features to enforce isolation at the machine word granularity, by virtually extending each memory unit with an additional tag that is defined by data-flow. To implement HDFI, we extended the RISC-V instruction set architecture and instantiated it on the Xilinx Zynq ZC706 evaluation board. Our evaluation using benchmarks including SPEC CINT 2000 showed that the performance overhead due to our hardware modification is low ($< 2\%$).

Chapter 5

Data Space Randomization

5.1 Motivations

Memory corruption remains an important attack vector in practice. Exploits of memory corruption vulnerabilities typically force programs into performing attacker-chosen operations, leaking secrets, or both. This security problem is being addressed from many angles including i) migration to type safe languages, ii) static and dynamic program analysis, and iii) retrofitting unsafe code with memory safety mechanisms. All of the above require a "developer-in-the-loop" whereas exploit mitigations are transparent to developers and end-users. Mitigations also avoid the substantial overheads associated with full memory safety enforcement [16, 145, 114].

Automatic mitigations have been highly effective at driving up the cost of exploitation. Today, techniques such as Address Space Layout Randomization (ASLR) [156], Data Execution Prevention (DEP) [157], Stack Canaries [158], and Control Flow Integrity (CFI) [159] are widely deployed in modern operating systems and compilers.

Current standard mitigations have a narrow aim: stopping code injection and code-reuse techniques which hijack the program execution. Adversaries are therefore shifting to data-oriented attacks [105, 106], which do not divert the control flow of the target program from its intended path. Data-oriented attacks use maliciously crafted payloads to corrupt the data flow of the program and were shown to be as powerful as traditional code execution attacks [160].

Data Space Randomization (DSR) is a promising defense which can effectively mitigate data-oriented attacks. DSR, which was independently and concurrently proposed by Bhatkar et al. [161] and Cadar et al. [162], seeks to thwart unintended data flows arising from memory corruption vulnerabilities while leaving all legitimate data flows unaffected. To do so, DSR encrypts (masks) variables that are stored in the protected program's memory, and it uses different keys, which are randomly chosen when the program starts, to encrypt unrelated variables. The protected program is instrumented accordingly: every memory store is preceded by an encryption operation, and every memory load is succeeded by a decryption operation. By choosing the encryption keys with sufficient entropy, DSR makes it hard for attackers to predict the results of store operations that violate the program's intended data flow, thereby making reliable construction of data-oriented attacks very difficult.

Existing implementations of DSR are far from perfect. They incur too much run-time overhead to meet the bar for widespread deployment [1], even when they trade off security for better run-time performance. They also rely on fast but imprecise points-to analysis which results in fewer equivalence classes being used to separate unrelated data accesses. Consequently, current DSR implementations do not affect many of the unintended data flows; attackers may use this leeway to construct exploits. Finally, current approaches store the encryption keys (masks) in attacker-observable memory, which makes DSR prone to information-leakage attacks.

This motivated our work on `HARD`, a practical and efficient hardware-assisted defense that addresses the shortcomings of previous DSR approaches. `HARD` consists of two major components: a novel extension of the RISC-V architecture, and a proof-of-concept implementation of DSR that leverages the capabilities of this extension. The architectural extension includes specialized instructions to store and load encrypted program data, and dedicated caches to maintain the keys used to encrypt this data.

The proof-of-concept implementation of DSR is inspired by previous work in the way that it encrypts data accesses. Unlike previous work, however, we use a more precise (context-sensitive) alias analysis which lets us dynamically determine which keys to use based on the current calling context. Using context-sensitive analysis lets us separate data accesses into more equivalence classes, which in turn increases security over prior work.

Our approach has several advantages. First, our architectural extension enables highly efficient masking of program data. By performing the encryption/decryption operations directly in hardware, and by maintaining the encryption keys in specialized caches, the run-time overhead of `HARD` was about 12% on average when we run SPEC CINT 2000 Benchmark with `train input`. This was about three times smaller when compared to a software-only implementation. Second, our extension can increase DSR's resilience against information-leakage attacks. Our keys are stored in a memory region which is not mapped to the protected program memory, and the processor never stores the keys into any register or on-chip memories which the code loads from or stores to. Therefore, an attacker that can access arbitrary program memory using a memory corruption vulnerability cannot use this vulnerability to disclose the keys. Third, our key management instructions allow us to dynamically assign encryption keys to memory accesses, which enables an efficient implementation of context-sensitive encryption. Fourth, by assigning the encryption keys based on context, we can increase the number of equivalence classes compared to existing DSR implementations, and, consequently, offer greater security. Finally, we designed our architectural extension to be generic enough to support a range of different DSR schemes allowing implementors to experiment with different compilers, types of points-to analysis and encryption schemes.

In summary, we contribute the following:

- We present `HARD`, a novel hardware-assisted defense against data-oriented attacks. `HARD` consists of an extension to the RISC-V architecture, including instructions to encrypt/decrypt data directly and caches to manage encryption keys, and a proof-of-concept implementation of context-sensitive Data Space Randomization (DSR). `HARD`'s architectural support can increase the efficiency and security of DSR schemes and can increase DSR's resilience against information leakage attacks. `HARD`'s DSR implementation leverages the capabilities of the architectural support and, by assigning encryption keys dynamically based on the calling context, it is able to offer strictly greater security than existing implementations of DSR.

- We implemented the architectural extension by modifying the *Rocket Chip Generator* [148], so that it generates a system with `HARD` enabled, and we instantiated the system on the Xilinx Zynq ZC702 evaluation board [163] which includes Xilinx Zynq Z-7020 [31]. We implemented the DSR scheme on top of version 3.8 of the LLVM Compiler Infrastructure [164] and the PoolAlloc module [165].
- We present a careful and detailed evaluation of `HARD`'s performance and security properties and show that it is more efficient and secure than existing DSR implementations.

5.2 Background

Our goal is to thwart attacks that violate the intended data flow of a program. We use an example to illustrate two types of violations: *use-after-free* and *uninitialized reads*. We chose these violations because use-after-free errors are known to be expensive to prevent with other techniques [166], yet are very relevant in practice [167], and an uninitialized read violation was used in the infamous Heartbleed vulnerability [168].

Example 5.1 illustrates both violations. At lines `(a-1)` and `(a-2)`, the program allocates and initializes a list, `X`, as depicted in Figure 5.1-(a).

At line `(b-1)`, the second element of list `X` is freed, so the `next` member of the first element becomes a dangling pointer. A new list, `Y`, is allocated at line `(b-2)`.

```c
typedef struct list { struct list *Next; int Data; } list;
list *makeList(int Num) {
  list *New = (list*)malloc(sizeof(list));
  if(!New) abort();
  New->Next = Num ? makeList(Num-1) : 0;
  return New;
}
void fillList(list* L, int base){
  if((L->Next)) fillList(L->Next,base+1);
  L->Data = base;
}
void delList(list* L){
  if((L->Next)) delList(L->Next);
  free(L);
}
void dumpList(list* L){
  list *pt = NULL;
  pt = L;
  while(1){
    printf("%p:\t%d\n",&(pt->Data),pt->Data);
    if(pt->Next == NULL) break;
    else pt = pt->Next; }
}
int main(int argc, char** argv){
  list *X = makeList(3);    // (a-1)
  fillList(X,10);           // (a-2)
  dumpList(X);
  free(X->Next);            // (b-1)
  list *Y = makeList(2);    // (b-2)
  dumpList(Y);              // (b-3)
  fillList(Y,20);           // (c-1)
  dumpList(X);              // (c-2)
  delList(Y);
  return 0;
}
```

Example 5.1: *A synthesized program illustrating use-after-free and uninitialized read vulnerabilities.*

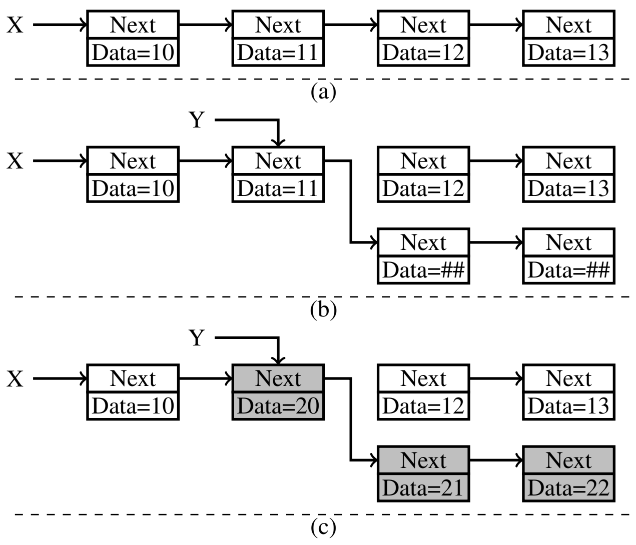

Figure 5.1: The diagram shows the lists generated in Example 5.1. (a) shows list `X` *after initialization at line* `(a-2)`*. (b) shows the most likely layouts of lists* `X` *and* `Y` *when the uninitialized read happens at line* `(b-3)`*. (c) shows the most likely layouts of the lists at line* `(c-2)`*, when the use-after-free happens.*

The contents of list `Y` are read without initialization at line `(b-3)`. Due to the deterministic nature of common memory allocators such as *dlmalloc* [169] in GNU's C standard library, *glibc*, the two lists will likely be laid out in the memory as shown in Figure 5.1-(b). Thus, the data read at line `(b-3)` will likely include the recently free'd element of list `X`.

While this particular example is contrived, similar uninitialized read vulnerabilities exist in practice. One was recently found in the widely-used OpenSSL library. This vulnerability was actively exploited and dubbed the *Heartbleed* bug [168, 170]. A scan of the Alexa Top 100 websites, run shortly after the discovery of the bug, showed that at least 44 of these websites were vulnerable [171].

The rest of the program demonstrates the use-after-free vulnerability. In C and C++, it is possible but not legal to access objects, like the list elements of `X` after they have been deallocated. Attackers may take advantage of such temporal memory safety violations, as we demonstrate at line `(c-2)`. The example program attempts to print the contents of list `X`, whose second element was freed at line `(b-1)`. With a deterministic memory allocator, the list `X` will likely be laid out as shown in Figure 5.1-(c), and the dumped list will likely include elements of list `Y`.

5.2.1 Mitigation with DSR

DSR can mitigate exploits such as the ones we just illustrated. The idea is to encrypt memory objects and to instrument the program code so that legitimate data flows remain unaffected. To this end, encryption operations are added before every memory store, and decryption operations are added after every memory load. To the extent possible, unrelated storage locations are encrypted using different keys.

Storage locations (e.g., objects) are either accessed by name or by reference, i.e., through pointers. We perform alias analysis to compute the points-to relation between pointers and the storage locations they can reference. Two pointers are said to be aliases if they can reference the same storage location. Similarly, a pointer p may alias named object o, if p can point to o. The aliasing $=_a$ relationship between two data references is defined as follows:

Definition 5.2.1 $r_1 =_a r_2$ *if there exists a valid program execution where, at some point,* r_1 *and* r_2 *reference the same storage location.*

Based on the alias analysis, DSR partitions storage locations into *equivalence classes*. All storage locations belong to an equivalence class, and any two storage locations that can be accessed by the same data reference (taking aliasing relationships into account) are in the same equivalence class. The equivalence class relationship is transitive. If a data reference r_1 can access locations l_1 and l_2, and another data reference r_2 may access l_2 and l_3, then l_1, l_2, and l_3 must be in the same equivalence class, even if there is no data reference that can access both l_1 and l_3.

The idea is that DSR can encrypt storage locations belonging to distinct equivalence

```
int main(int argc, char** argv){
  int *X = malloc(3 * sizeof(int));
  if(!X) abort();
  for (int i = 0; i < 3; ++i) X[i] = i;

  int *Y = malloc(2 * sizeof(int));
  if(!Y) abort();
  for (int i = 0; i < 2; ++i) Y[i] = i * 2;

  for (int i = 0; i < 3; ++i) printf("%d\n", X[i]);
  return 0;
}
```

Example 5.2: *A simple program in which existing DSR schemes achieve full precision.*

classes with distinct encryption keys without affecting correctness in the absence of memory corruption. Locations belonging to the same equivalence class, however, must be encrypted with the same key. If we apply this core idea to the program in Example 5.2, we will find that arrays `X` and `Y` never alias, which means that DSR's alias analysis will place the arrays into different equivalence classes. DSR will therefore encrypt the arrays with different encryption keys, and the `printf` call will use array `X`'s encryption key to array elements prior to printing them.

5.2.2 Limitations of Existing DSR Schemes

While all existing DSR implementations are able to mitigate the vulnerabilities in the simple program in Example 5.2, they cannot mitigate the vulnerabilities in the earlier Example 5.1. The underlying problem is that previous DSR implementations used *context-insensitive* alias analysis which computes a single solution valid at any point in the program. This leads to imprecision because the aliasing relations inside a function change depending on where the function was called, i.e., the calling context. We avoid this loss of precision by using a *context sensitive* alias analysis. We define a context-sensitive aliasing relationship $=_{a[c]}$ as follows:

Definition 5.2.2 $r_1 =_{a[c]} r_2$ *if there exists a valid execution where, in calling context c,* r_1 *and* r_2 *reference the same storage location.*

In the earlier program, both `X` (at line `(a-2)`) and `Y` (at line `(c-1)`) are passed as arguments to `fillList`, so a *context insensitive* alias analysis will report that the formal argument `L` of `fillList` may alias both `X` and `Y` cf. Definition 5.2.1. Variables `X` and `Y` will therefore be assigned to the same equivalence class. Similar aliasing relationships are introduced by the arguments passed to `dumpList`, and by the return values returned from `makeList`.

If we analyze our example program with a context-sensitive alias analysis, we obtain two sets of aliasing relations: one for the calling context at line `(a-2)` where `fillList`'s formal argument `L` aliases `X`, and one for the calling context at line `(c-1)` where `L` aliases `Y`. By taking the calling context into account, we avoid having to treat `X` and `Y` as aliases and can therefore place them in different equivalence classes.

In summary, we can construct a higher number of equivalence classes, that are smaller in size (and thus more precise) relative to previous DSR techniques by taking context into account when computing aliasing relations. In Example 5.1, lists `X` and `Y` alias according to definition 5.2.1, but do not according to definition 5.2.2.

Leveraging the greater precision of context-sensitive alias analyses is challenging since the DSR instrumentation code must then take the calling context into account to determine which encryption key should be used. We discuss this challenge at length in Section 5.4, and present a novel DSR scheme that supports different contexts via dynamic key binding.

5.3 Threat Model

Our work uses the following threat model which is realistic and consistent with related work in this area [160, 17]:

- The victim program contains a memory corruption vulnerability that lets adversaries read and write arbitrary program storage locations as long as such accesses are allowed by the permission bits in the page table.

- We assume that the victim program is protected against code injection by enforcing an $W \oplus X$ policy such that no executable code is writable and vice versa.
- We assume that the victim program is protected against code reuse by enforcing a fine-grained, C++ aware control-flow integrity [83, 172] on forward edges and using a shadow stack [74] to protect backwards edges (returns). This leaves adversaries with the option to mount data-oriented attacks.
- We assume that the victim program runs in user mode and that the host system's software running in supervisor mode has not been compromised by the adversary.
- We do not consider side-channel attacks, flaws in the hardware, or adversaries that have physical access to the system hosting the victim program.

5.4 Design

Before we discuss the design of `HARD` and the DSR implementation we built on top of it, we provide a general model of the DSR implementations supported by `HARD`. Any DSR implementation, or any general memory encryption scheme that can be made to fit this model, can be accelerated by `HARD`.

As described in Section 5.2, DSR uses alias analysis to cluster the set of memory locations in a given program into equivalence classes. Previous implementations of DSR build these equivalence classes based on a context-insensitive alias analysis. We aim higher, and want to support not only the existing DSR schemes, but also improved DSR schemes based on context-sensitive analysis.

A context-insensitive alias analysis does not take the calling context into account when assigning encryption keys to equivalence classes. This means that the DSR scheme can instrument each memory access instruction with encryption/decryption operations

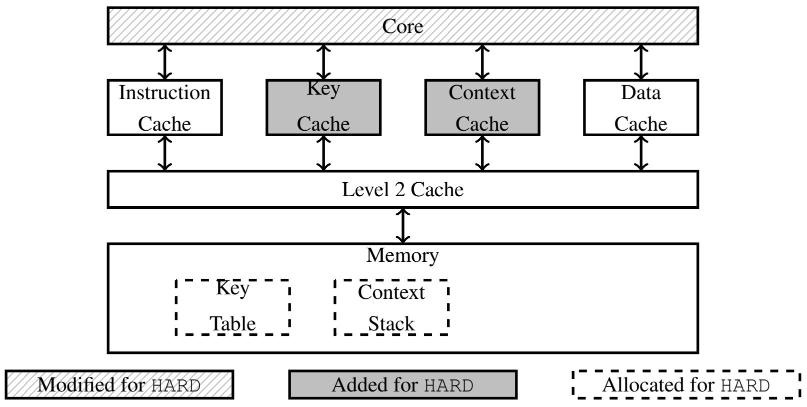

Figure 5.2: Hardware overview for a `HARD`-enabled system.

that use *statically* assigned keys, based on which equivalence class the memory access instruction refers to. A context-sensitive alias analysis does take the calling context into account. This means that the DSR scheme must select the encryption key used for each memory access instruction *dynamically*, based on the context that instruction executes under. We designed `HARD` to support both models, and we therefore give DSR/memory encryption schemes the freedom to choose between static and dynamic key assignment on a per-instruction basis.

We only target DSR/memory encryption schemes that use `xor` masking for encryption and decryption. This too is compatible with existing DSR implementations. Our design supports context-sensitive DSR implementations, but does not identify the context automatically. Rather, we leave it up to each DSR implementation to identify and manage the context.

5.4.1 Hardware Overview

`HARD` adds or modifies several hardware components in support of DSR. Figure 5.2 shows a general overview of a `HARD`-enabled system. When executing a `HARD`-enabled program, the processor uses two reserved memory regions, the *Context Stack* and the *Key Table*, to store and manage the encryption keys used in the program. These memory regions do not have to be mapped into the virtual address space of the protected program, as the processor accesses the regions directly using their physical addresses. This design allows us to prevent the encryption keys from leaking as an adversary cannot read physical memory directly in our threat model.

The Key Table stores all of the encryption keys used in the program: both the ones that are statically assigned, and those that are dynamically assigned. To support dynamically assigned keys, programs must create context frames on the context stack and copy the keys used within a context from the Key Table to the context frames.

Both regions have a corresponding cache: the *Context Cache* and the *Key Cache* respectively. Similar to the memory regions, the caches cannot be read directly by an attacker. Dynamically assigned keys are always loaded via the Context Cache. If the key is not present in the cache, the processor will transparently fetch it from the corresponding Context Stack. Similarly, statically assigned keys are always loaded via the Key Cache.

To pair the caches with their corresponding memory regions, we added a base register to each cache. This base register contains the physical memory address of the associated memory region.

5.4.2 Hardware Initialization

The OS kernel is responsible for the initialization of the aforementioned memory regions and caches. When the OS loads a `HARD`'ened program, the kernel allocates the Key Table and initializes it with randomly generated encryption keys. The kernel then sets the base address register of the Key Cache and activates it using a control register. The Key Cache is automatically flushed when it is activated.

The kernel then allocates the Context Stack and sets the base address register of the Context Cache. Unlike the Key Table, it is not necessary to initialize the Context Stack, as `HARD` requires the program to explicitly manage the Context Stack at run time.

5.4.3 New Instructions

`HARD` adds two sets of instructions. One set of instructions is used to load data from or store data to encrypted memory. The other set of instructions is used to manage the Context Cache and Context Stack.

Memory Access Instructions The RISC-V instruction set architecture, which we extend, contains seven load instructions and four store instructions. For each of these, `HARD` adds a specialized version that decrypts data when loading or encrypts data when storing. The specialized instructions use the same mnemonic as the instructions they are based on, but have a `um` suffix (for loads) or `m` suffix (for stores). The instructions to load and store double words, for example, look as follows:

- `ldum rd, id(rb)`: load a double word from the virtual address stored in register `rb`, decrypt the data with the key at index `id` in the Key Table/Context Stack, and write the decrypted data to register `rd`.
- `sdm rd, id(rb)`: encrypt the data in register `rd` with the key at index `id` in the Key Table/Context Stack and store the encrypted data to the virtual address in register `rb` as a double word.

The type of encryption key is encoded in the Most Significant Bit (MSB) of the index id. If the MSB is set to 0, the remainder of the index id is interpreted as an index into the Key Table, and the instruction therefore has a statically assigned key. We refer to these index IDs as *static IDs*. If the MSB is set to 1, the remainder of the index id is interpreted as an index into the current context frame on the Context Stack, and the instruction therefore has a dynamically assigned key. We refer to these index IDs as *dynamic IDs*.

Context Cache Management Instructions The second group of the instructions are used to manage the Context Stack. Before a new context is entered, the program should prepare a new context frame on the Context Stack, and copy the keys used within the corresponding context into that frame. This newly prepared context frame must then be activated before entering the corresponding context. `HARD` offers four instructions, as illustrated in Figure 5.3, to prepare, activate, and deactivate context frames.

- `mksc dest_id, src_id`: move the key at index `src_id` in the Key Table to slot `dest_id` in the context frame under preparation.
- `mkcc dest_id, src_id`: move a key from slot `src_id` of the currently activated context frame to slot `dest_id` of the context frame under preparation.
- `drpush cur_len`: deactivate the active context frame and activate the context frame under preparation. `cur_len` represents the number of slots in the current.
- `drpop`: deactivate the active context frame and activate the previous context frame.

Although it is not required, context frames are usually tied to calling contexts. The `mksc`, `mkcc`, and `drpush` instructions are therefore meant to be used just before calling a function, while the `drpop` instruction is meant to be used just before the function returns.

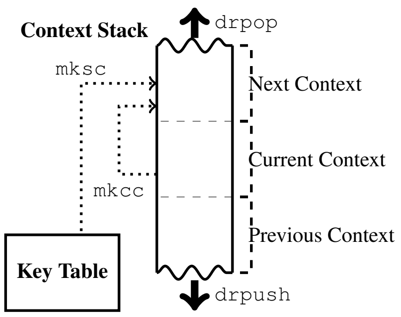

Figure 5.3: `HARD`'s Context Stack management instructions. `mksc` *copies a key from Key Table to an entry of the next context.* `mkcc` *copies a key from the current context to the new context.* `drpush` *and* `drpop` *deactivate the current context frame and shift the stack to activate the next or previous context frame respectively.*

5.4.4 DSR Overview

We designed a DSR scheme that leverages the capabilities of `HARD`'s architectural extensions. Our DSR scheme is applied in four steps. In the first step, we run a context-sensitive alias analysis and cluster the program's memory locations into equivalence classes based on the results of the alias analysis.

In the second step, we mark the original memory access instructions in the program based on the equivalence classes they refer to. We mark an instruction as `static` if the equivalence class it refers to is the same regardless of the calling context. If the equivalence class an instruction refers depends on the calling context, we mark that instruction as `dynamic`.

In the third step, we embed the context management instructions described in Section 5.4.3. These instructions are added to every function that calls a function containing memory access instructions marked as dynamic. This step adds the necessary instructions to prepare context frames containing the keys for all of the equivalence classes referred to by instructions we marked as dynamic in the previous step.

In the final step, we replace the marked memory access instructions with the specialized instructions that decrypt while loading and encrypt while storing. For instructions marked as static, we can refer to the encryption key directly, using that key's index in the Key Table. For instructions marked as dynamic, we refer to the encryption key's assigned slot in the context frame.

5.5 Prototype Implementation

We implemented our proposed hardware architecture by extending the Rocket Chip Generator [148]. Specifically, we extended one of the instances it can generate. This instance is composed of the Rocket Core [173], a 16KiB L1 instruction cache, a 16KiB L1 data cache, and a 256KiB unified L2 cache. The Rocket Core has a 6-stage pipeline, and its performance has been reported to be similar to that of ARM Cortex-A5

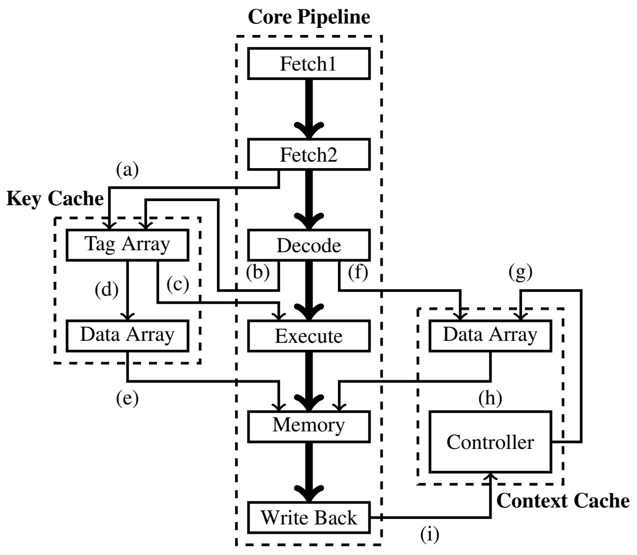

Figure 5.4: Overview of the modified Rocket core, showing the interaction between the original core pipeline and the Key and Context Caches added by `HARD`*.*

processor [174]. We added the two hardware components, Key Cache and Context Cache, to the system with the Rocket Core. The core pipeline is also augmented to interact with the additional components. Figure 5.4 gives an overview of the interaction between the components and the core pipeline, which will be described further in this section.

5.5.1 Instruction Encoding

The way we encode the specialized memory access instructions described in Section 5.4.3 is an important design parameter for `HARD` as it determines the upper bound of the Key Table size and the Context Stack size. To avoid intrusive changes to the existing instruction decoder, we designed our specialized instructions so they have an encoding similar to the instructions they are based on. Our specialized instructions differ from these base instructions in only one respect: they interpret their immediate fields as index IDs, rather than memory offsets. This means that, like these memory offsets, the size of the index IDs is limited to twelve bits. We use the most significant bit of the index ID to indicate whether the index should be interpreted as an index into the Key Table, or an index into the current frame on the Context Stack. This leaves us with eleven bits to encode the ID itself.

Two of our context management instructions, `mksc` and `mkcc`, require two index ID operands, a source ID and a destination ID. For this reason, we based these instructions on the existing instruction that can encode the longest immediate field, which is 20 bits long. The semantics of our context management instructions define the type of index IDs they operate on, so we do not have to encode this type in the MSB of the index ID immediates. The `mksc` instruction operates on a static ID (index into the Key Table) and a dynamic ID (index into the current context frame), and the `mkcc` instruction operates on two dynamic IDs. The size of these pairs of index IDs cannot exceed the 20 bits that are available to encode them.

We analyzed a large number of programs and found that 512 is a realistic limit to the number of dynamic keys that can be loaded into a single context frame. We therefore chose to limit the size of dynamic IDs to 9 bits, and the size of static IDs to 11 bits. The size of the Key Table is, in other words, limited to 2048 entries, whereas the size of context frames is limited to 512 slots. To support functions containing more than 512 memory access instructions that access different equivalence classes using dynamically assigned keys, we therefore require that some of these equivalence classes use the same encryption keys.

Our other context management instructions, `drpush` and `drpop`, are pseudo-instructions that are not actually implemented in the hardware. Rather, the RISC-V assembler we modified translates these instructions into system control instructions that generate Context Cache commands through the Control and Status Registers (CSR) interface.

5.5.2 Processor Pipeline

We modified the Rocket Core's 6-stage core pipeline to enable interaction with the Key and Context Caches, as shown in Figure 5.4. The modified pipeline can send static IDs to the Key Cache, which responds with the corresponding statically assigned encryption key. It can also send dynamic IDs to the Context Cache, which responds with the corresponding dynamically assigned encryption key.

5.5.2.1 Key Cache

The Key Cache is a fully-associative cache that services requests for static IDs by loading the corresponding keys either from its data array or from the Key Table in the memory. The Key Cache is composed of two components, which are the pipeline depicted in Figure 5.4, and a miss handler.

The cache has a tag array, containing a set of $(valid, id, offset)$ tuples. For static IDs whose data is currently present in the Key Cache's data array, this tuple gives us the offset of that static ID's corresponding encryption key in the data array. The data array has a size of 1KiB, which means that it can contain 128 keys. If the core pipeline requests a key that is not currently present in the data array, the miss handler will load that key directly from the Key Table. Note that keys that are evicted from the data array do not need to be written back to the memory, as the Key Table cannot be modified at run time.

Because of the tag array access and tag matching, our Key Cache takes two cycles to respond to a request, even if the requested key is present in the data array. The Execute stage of the Rocket Core is only one cycle long, however. If we would request the keys from the Decode stage, we would therefore stall the Execute stage of any of our specialized memory access instructions for one cycle, even if the requested key is in the cache. To avoid this unnecessary stalling, we forward the raw instruction bytes fetched by the Fetch2 stage to the Key Cache. The Key Cache then decodes the instruction using a minimal decoder to determine if the forwarded instruction contains a static

ID. If the instruction does indeed contain a static ID, the Key Cache will look up the corresponding key immediately, rather than wait for the request from core pipeline's Decoder step. This way, the Key Cache provide the Execute stage with the appropriate key without stalling, if the key was present in the cache, or it will indicate that the key needs to be fetched from memory.

5.5.2.2 Context Cache

The Context Cache consists of two major components, the stack controller and the data array, following the design of a hardware stack with on-chip memory presented in earlier work [175]. The cache has dedicated registers to keep of the location of three context frames: the previous frame, the current (activated) frame, and the next frame. Whenever the program executes a `drpush` and `drpop` instruction to activate a new frame, the stack controller updates the dedicated registers accordingly. If the available space in the data array exceeds certain limits after executing one of these instructions, the cache might evict the oldest entries to the in-memory context stack to make room for new entries, or fetch old entries from the in-memory stack because enough space is available.

The Context Cache's data array is 1KiB big, which, in our design, is sufficient to store the top 128 slots on the Context Stack. When the program copies an encryption key to the Context Stack using the `mkcc` or `mksc` instructions, the key will be stored directly in the corresponding slot of the next frame in the Cache's data array.

The eviction, fetching of the data array entries and the changes of the context frame registers of the Context Cache happen at the last stage of the pipeline, which creates a possible hazard for other instructions in the pipeline, that might also access the Context Cache. We therefore modified the pipeline so that, whenever a `drpush` or `drpop` instruction is decoded, or whenever an eviction or fetch operation is in progress, any succeeding instructions that access the context cache are stalled until the `drpush`/`drpop` has finished executing, or the eviction/fetch operation has completed.

5.5.3 DSR prototype

In addition to the `HARD` architectural extensions, we also implemented a fully-fledged DSR prototype which is able to make use of the `HARD` instructions. Our implementation is based on LLVM/Clang 3.8 [164] for RISC-V [128], and uses alias analysis algorithms from the LLVM PoolAlloc module [165]. Our implementation of DSR is the first to make use of a context-sensitive analysis and to dynamically select the key used for memory accesses based on the program's calling context at run time. Our implementation takes full advantage of the capabilities of `HARD` and the performance and security benefits it provides.

To evaluate the performance benefits of `HARD` we also implemented a software-only version of our DSR scheme, which only uses instructions from the base ISA. To make the comparison fair, the same alias analysis results are used for the `HARD` and the software-only implementations. The set of equivalence classes used in each implementation is identical.

Our software-only implementation is not only slower than the `HARD`-enabled implementation, but also suffers from the same security limitations as all pre-existing DSR implementations: some keys are encoded directly into instructions as immediate operands, while other keys are stored in readable data regions. Both types of keys can be leaked through information disclosure attacks [176, 90, 177, 178].

5.5.3.1 Class Assignment

We use Data Structure Analysis (DSA) [179] to categorize the memory objects into equivalence classes. DSA is a context-sensitive alias analysis that scales well to large programs. The output of DSA is a points-to graph for each function, which incorporates the aliasing effects of all callers and all callees of that function. For each function and its associated points-to graph, we assign equivalence classes based on Bhaktar and Sekar's mask assignment algorithm [161], with a slight modification to differentiate the static and dynamic equivalence classes. A node in the points-to graph represents a set of memory objects joined through aliasing relationships, and each node represents a disjoint set of objects. Therefore each node identifies a distinct equivalence class. For each node and associated equivalence class, we select either a static or a dynamic ID as follows: If the node contains or is reachable from any of the function's arguments or its return value, then that equivalence class is considered dynamic and we assign it a dynamic ID. In all other cases we assign the equivalence class a static ID.

5.5.3.2 Program Transformation

We implement our DSR transformation as an LLVM link-time optimization pass. This makes the entire program's code visible to the alias analysis. We perform our program transformation after DSA is performed and after we have formed the equivalence classes.

Our transformation pass starts by annotating load and store operations at the LLVM Intermediate Representation (IR) level. The annotation contains the class ID assigned to the memory location these operations access.

We then insert the instructions to manage the context stack. For each call site, we construct a mapping from the class IDs of the actual arguments in the caller context to the dynamic IDs of the formal arguments of the callee function. This mapping is used to insert `mksc` and `mkcc` instructions which initialize the callee context. The `drpush` instruction is inserted directly before call instructions to switch to the callee's context, and `drpop` is inserted before return instructions to restore the caller's context. Finally, during code generation the annotated loads and stores are emitted as the specialized instructions with the dynamic or static equivalence class ID encoded into the immediate operand.

The software-only version performs the same tasks without using our new instructions. The first steps, which construct the equivalence classes and annotate the instructions, are done exactly as described above. However, since there is no Context Stack, we track the context using the program's stack, and pass the key values for dynamic equivalence classes as additional function arguments.

The software-only version also does not have a Key Table, so it uses the key values directly. Each static ID is bound to a unique 8-byte key and each dynamic ID is bound to an additional formal argument. Each unique dynamic ID within a function causes an extra formal argument to be added, and calls are modified to track the context by passing the correct key values in the additional arguments. In the final step, we insert additional instructions to explicitly perform the XOR encryption before stores and decryption after loads using the key values assigned to the class IDs.

5.5.3.3 Support for external binaries

The DSR analysis and transformation requires that the source code of the program is available in order to classify data into different equivalence classes and instrument the memory accesses. If the source code for a portion of a program is not available, for example, if the program calls functions in an external library, then we follow the same approach as prior DSR implementations and route those function calls through wrapper functions we create [162, 161]. The wrapper functions handle the transition from instrumented to non-instrumented code by decrypting the memory reachable from the arguments of the original function call before calling the original function. When the original function returns, the wrapper re-encrypts the arguments and return values before returning to the instrumented code. For our implementation of DSR we implemented wrapper functions manually, for all 71 C library functions used in SPEC CINT 2000. Implementing new wrappers is straightforward, and can be mostly automated.

5.6 Security Evaluation

Like other DSR implementations, `HARD` is designed to stop illegitimate data flows. Illegitimate data flows are possible if an attacker can force a memory store instruction

Table 5.1: Number of static equivalence classes obtained for a context-insensitive and context-sensitive analysis.

Benchmark	**Context Insensitive**	**Context Sensitive**
164.gzip	127	145 (14.17%)
175.vpr	718	785 (9.33%)
176.gcc	2150	2985 (38.84%)
181.mcf	41	41 (0.00%)
186.crafty	1130	1160 (2.65%)
197.parser	343	443 (29.15%)
252.eon	1558	1676 (7.57%)
253.perlbmk	492	527 (7.11%)
254.gap	395	500 (26.58%)
255.vortex	916	1505 (64.30%)
256.bzip2	96	106 (10.42%)
300.twolf	694	800 (15.27%)
geomean		(17.62%)

to write to an arbitrary memory location, that is later read by load instructions in the program. `HARD` is able to stop such attacks, but only if it has statically determined that the store and load instructions target different equivalence classes. If the load and store target the same equivalence class, the attack will not be stopped. This also applies to Data-Flow Integrity [15], which also relies on static analysis.

Since DSR will only stop an attack that overwrites data from another equivalence class, having many fine grained equivalence classes is important for security. Therefore, the total number of equivalence classes for a given program is a meaningful metric in evaluating the security of a DSR implementation. To compare our approach, which uses a context-sensitive analysis to build equivalence classes, with previous work, which used a context-insensitive analysis, we implemented a version of `HARD` based on Steensgaard's Analysis [180], a context-insensitive alias analysis that was used by Bhatkar et al. in their DSR implementation [161]. We compiled the SPEC CINT 2000 benchmarks and counted the number of equivalence classes created by both the context-sensitive and the context-insensitive versions of `HARD`. These results are shown in Table 5.1.

The average increase in the number of equivalence classes when using a context-sensitive analysis is 17.62%. The context-sensitive version of `HARD` therefore enforces a stricter data-flow policy than the existing DSR schemes.

As we pointed out in Section 5.5.1, `HARD`'s hardware limits the size of static IDs to 11 bits, which, in turn, limits the number of equivalence classes to which we can assign distinct encryption keys to 2048. One benchmark, `176.gcc`, exceeds this number of equivalence classes. To run `176.gcc`, as well as other program with over 2048 equivalence classes, we are therefore forced to assign some static IDs to multiple equivalence classes. Consequently, `HARD` enforces a less precise data-flow policy in such programs. The security impact of static ID reuse could be reduced by carefully choosing which equivalence classes may share IDs. For example, if we could statically determine that two equivalence classes do not contain security-critical memory locations, then it should be safe to assign them to the same static ID. Statically finding security-critical variables is a difficult problem, which we leave for future work.

Besides the increased number of equivalence classes, `HARD` has an additional advantage over prior work. Both of the existing DSR schemes contain an optimization that significantly reduces the run-time overhead, in return for lowered security. This optimization disables encryption for equivalence classes that are statically determined to be "safe". An equivalence class is considered safe if a static analysis can assert that none of the accesses to that equivalence class can read or write outside the bounds of the target object. This reduces the security because, even if an object can not overflow, it could still be the target of a use-after-free or uninitialized-read vulnerability. `HARD`, does not implement this optimization, and encrypts all equivalence classes, thereby providing protection against these additional attacks.

5.6.1 Real-World Protection

We also investigated the ability of `HARD` to defend against a real world exploit. Unfortunately, we could not find any relevant exploits for program binaries running on the RISC-V architecture, which we target with `HARD`. For this reason, we ported our software-only implementation of DSR to the x86 architecture, because several documented exploits are available for x86 programs. Our software-only implementation provides the same level of protection against data-oriented exploits on both platforms.

We evaluated `HARD`'s DSR implementation against a recent data-oriented attack presented by Hu et al. [106]. The exploit uses a format string vulnerability in the *wu-ftpd* server to perform privilege escalation. This is a strong attack in which an adversary can read and write to arbitrary memory locations. In this exploit, the attacker overwrites a global pointer to a `struct passwd`. The overwritten pointer is then dereferenced by the server, and the dereferenced value is interpreted as a user ID. This user ID is subsequently used as an argument for a `setuid` call. By overwriting the global pointer with the address of a memory location that contains a value of 0, which is the user ID of the privileged root user, an attacker can therefore escalate the privileges of the vulnerable application.

We compiled two versions of the *wu-ftpd* binary: a base version, and a `HARD`'ened version. In the base version, we only enabled Control-Flow Integrity (CFI) and Stack Canaries. In the `HARD`'ened, we enabled `HARD`, as well as the aforementioned defense mechanisms. We then tested the exploit against both versions, on a system with Address Space Layout Randomization and Data Execution Prevention enabled. The exploit was successful against the base version, as the attack does not affect the control flow of the application in any way. The exploit did not work against the `HARD`'ened version, however. While the attacker is still able to overwrite the pointer in the `HARD`'ened version, the subsequent dereference operation used a different encryption key than the instruction that overwrote the pointer, making it impossible for the attacker to reliably control the outcome of the overwrite. This causes the `setuid` call to be called with the user ID to be set to an unpredictable value.

We examined the exploit further to determine if it could be adapted to defeat HARD. The data flow policy of HARD identifies three equivalence classes involved in this exploit: the class accessed by the vulnerable instruction during valid executions, the class of the pointer variable, and the class used for dereferences of the pointer. These classes are accessed using distinct keys, k_v, k_p, and k_d respectively. To reliably control the result of this exploit an attacker would have to learn two 64-bit secret values, $k_v \oplus k_p$ and k_d. If the attacker could discover these values then they could control the result of the pointer overwrite and could predict the result of the pointer dereference. However the keys used by HARD never appear in the address space of the program or in any general purpose registers, so they are not vulnerable to information disclosure attacks. The exploit thus requires an attacker to guess two 64-bit secrets, and therefore has a low chance of succeeding.

5.7 Performance Evaluation

To evaluate the run-time performance of HARD, we ran SPEC CINT 2000 [130] and compared HARD to baseline applications and the software-only (SWOnly) version of our DSR implementation. Although SPEC CINT 2000 has been deprecated in favor of SPEC CINT 2006, we chose to run the older version as our prototype system has limited resources.

Our prototype was instantiated using the *Rocket Chip Generator* [148] on the Xilinx Zynq ZC702 evaluation board [163]. Both the baseline system and HARD are run on the FPGA at 25MHz with 256MiB DDR3 memory. We used the RISC-V port of Linux kernel 4.1.17 modified to initialize the Key Table and Context Stack on program startup. We built all three versions of the benchmark binaries with LLVM/Clang 3.8 with link-time optimization enabled.

We could not run all of the benchmarks on their reference input sets, most likely

Table 5.2: Run-time overhead of `HARD` *and software-only DSR on SPEC CINT 2000.*

Benchmark	**Baseline**	**SWOnly**	**HARD**
		Train	
164.gzip	2569s	4447s (73.07%)	2970s (15.60%)
175.vpr	2092s	3383s (61.73%)	2786s (33.18%)
176.gcc	269s	438s (62.64%)	381s (41.54%)
181.mcf	1625s	1753s (7.91%)	1686s (3.80%)
186.crafty	1904s	3236s (69.92%)	2292s (20.35%)
197.parser	653s	810s (24.05%)	707s (8.35%)
252.eon	610s	750s (22.92%)	692s (13.45%)
253.perlbmk	3938s	4945s (25.57%)	4037s (2.52%)
254.gap	521s	625s (19.91%)	545s (4.51%)
255.vortex	760s	1069s (40.63%)	869s (14.31%)
256.bzip2	2184s	4047s (85.28%)	2704s (23.80%)
300.twolf	918s	1406s (53.23%)	1000s (8.91%)
geomean6		(27.62%)	(9.21%)
geomean8		(34.51%)	(10.33%)
geomean12		(37.64%)	(11.68%)
		Ref	
164.gzip	22198s	37137s (67.30%)	25512s (14.93%)
181.mcf	13381s	14315s (6.98%)	13755s (2.80%)
186.crafty	13811s	23634s (71.13%)	16655s (20.59%)
197.parser	27078s	33116s (22.30%)	29071s (7.36%)
252.eon	10672s	13187s (23.57%)	12170s (14.04%)
254.gap	13283s	- -	13356s (0.55%)
256.bzip2	13699s	22545s (64.57%)	15284s (11.57%)
300.twolf	27242s	- -	29169s (7.07%)
geomean6		(32.29%)	(10.05%)
geomean8		-	(6.69%)

because of the limited memory size of our system. We therefore report several sets of results. **Set 1** contains the six benchmarks that we could run on the reference inputs with each of the binaries we compiled. **Set 2** contains the eight benchmarks we could run on the reference inputs using the baseline version and the `HARD` hardware-assisted version, but not the `HARD` software-only version. **Set 3** contains all twelve benchmarks, running on their training inputs. We had to eliminate one of the input files in `253.perlbmk`'s training input set, as this file caused a crash.

Table 5.2 shows the results. We report several geometric mean overheads: geomean6 includes the six benchmarks from Set 1, geomean8 includes the eight benchmarks

Table 5.3: The number of L2 cache accesses while running SPEC CINT 2000 with `HARD` and software-only DSR.

Benchmark	Baseline	SWOnly	HARD
Train			
164.gzip	1.29M	1.36M (5.92%)	1.33M (3.61%)
175.vpr	1.23M	1.40M (14.04%)	1.29M (4.75%)
176.gcc	0.21M	0.35M (67.02%)	0.29M (40.66%)
181.mcf	1.36M	1.37M (0.66%)	1.37M (0.58%)
186.crafty	1.51M	3.08M (104.14%)	1.85M (22.85%)
197.parser	0.30M	0.30M (0.77%)	0.30M (0.07%)
252.eon	0.23M	0.35M (54.12%)	0.29M (24.91%)
253.perlbmk	2.46M	3.12M (26.66%)	2.54M (3.36%)
254.gap	0.22M	0.27M (21.04%)	0.23M (3.46%)
255.vortex	0.28M	0.48M (73.38%)	0.31M (10.31%)
256.bzip2	0.57M	0.58M (1.29%)	0.57M (0.98%)
300.twolf	0.64M	0.68M (6.15%)	0.64M (0.69%)
geomean6		(5.28%)	(2.07%)
geomean8		(6.40%)	(1.93%)
geomean12		(11.47%)	(3.22%)
Ref			
164.gzip	13.90M	14.50M (4.18%)	14.30M (2.84%)
181.mcf	10.80M	10.80M (0.12%)	10.80M (0.10%)
186.crafty	6.71M	9.94M (48.20%)	8.08M (20.45%)
197.parser	12.70M	12.80M (0.64%)	12.80M (0.75%)
252.eon	2.03M	3.12M (53.86%)	2.64M (30.01%)
254.gap	3.29M	- -	3.36M (2.34%)
256.bzip2	5.41M	5.51M (1.80%)	5.49M (1.32%)
300.twolf	16.60M	- -	17.40M (4.37%)
geomean6		(3.40%)	(2.37%)
geomean8		-	(2.55%)

from Set 2, and geomean12 includes all benchmarks from Set 3. The geometric mean overhead ranges from 6.69% to 11.68%, which is low compared to existing mechanisms for protecting against data-oriented attacks. The performance overhead of the hardware-assisted was about three times lower than that of the software-only version in all cases, showing the effectiveness of our architectural support.

`176.gcc` exhibits the highest runtime overhead of all of the SPEC CINT 2000 benchmarks. This is because of the number of keys needed for that application and the limited size of the Key Cache in our prototype. We used a Key Cache of 1KiB, which is small compared to the 16KiB instruction cache and data cache. We expect that the performance overhead could be reduced if we use a larger Key Cache which would lower the number of Key Cache misses.

The performance benefits of `HARD`'s hardware primarily comes from reduced cache misses for key values. To quantify the size of this effect, we measured the number of L2 cache accesses generated from the L1 caches, including the two new modules introduced by `HARD`.

Table 5.3 shows these results. In all cases `HARD` effectively reduced the number of L2 cache accesses.

5.8 Limitations

Known Plaintext Attacks Like the other DSR implementations, `HARD` is vulnerable against known plaintext attack because it uses `xor` operations with fixed keys to encrypt or decrypt data. If an attacker can disclose encrypted data and knows or is able to discover the plaintext data, then they can also recover the key used to encrypt that data. Once the key is known, the attacker can use it to craft the exploit payload such that the exploit will succeed. However, to reliably disclose data, the attacker must know the data layout of the target program. Randomizing this data layout can therefore mitigate this attack vector.

Attacks on Skewed Values Another attack vector against DSR is to target variables that have highly skewed values. These are variables where the range of valid values is a small subset of the possible values for the data type. An example is Boolean variables in C programs. In C, Boolean variables are integer-typed values where 0 is interpreted as `false` and all other values are interpreted at `true`. When the Boolean variable is written as `byte` to memory, the encrypted value can have 2^8 different values, but only one of them will be interpreted as `false`. If an attacker wishes to overwrite a Boolean variable to change a `false` value to `true`, the attack will have a very high probability

Table 5.4: Number of functions requiring each number of dynamic keys.

Dyn. Keys	0	1	2	3	4	5	6	7	8	9	10	>10
Functions	2729	377	215	90	52	23	12	10	7	5	2	5

of succeeding. Although any non-zero value is interpreted as `true`, in practice, many C programs are written such that Boolean variables will only have a limited number of values, often just 0 or 1. Attacks targeting these values could be mitigated by using a range analysis to identify the valid range of values and inserting checks to ensure that the plaintext data is always within the allowed range.

5.9 Future Work

While we have shown that our prototype can significantly reduce the performance cost of DSR, we believe that there are some opportunities for further optimization.

ID Width Reduction Since our specialized memory access instructions do not have an address offset field, protected programs need to have some additional arithmetic instructions to generate indexed addresses. A way to reduce the number of additional instructions is to reduce the width of IDs encoded in our specialized instructions. One way to do this is to make the ID width configurable. As shown in Section 5.6, many programs do not need 11-bit long static IDs, and Table 5.4 shows that they also do not need 9-bit long dynamic IDs. We could change our design so that each protected program could configure the core with the length of IDs that it needs and then can use the rest of bits as address offsets. For example, `164.gzip` can run with 8-bit static keys, which is sufficient to represent 256 keys. This would free the remaining three bits to encode address offsets. Another option is to partition the Key Table into multiple smaller tables and add an instruction that switches the active Key Table. If `HARD` uses four Key Tables for a program, the same total number of keys can be accessed using 9-bits, and the instructions will be able to use the remaining two bits to encode address offsets. The compiler could then minimize the number of table switching instructions by optimizing key assignments for temporal locality. Keys that are used in close proximity of each other could be assigned to the same Key Table.

Lazy Key Assignment for Loads While the current implementation makes the key for an instruction available when it arrives at the Memory stage, this is actually only required for store instructions. Since the load instructions will decrypt the data and write it back to registers during the Write Back stage, it is sufficient to make the keys available then. This would especially help performance when both a data cache miss and the Key Cache miss occur. In this case the latency of acquiring keys can be hidden as both the data cache and Key Cache can generate L2 Cache access in parallel.

5.10 Related Work

The idea of Data Space Randomization (DSR) is not new [161, 162]. However, the existing schemes only randomize the equivalence class containing so-called *overflow candidates*, because the performance cost of randomizing all classes is unacceptable. Existing schemes also use a context-insensitive alias analysis. `HARD` provides greater security by using context-sensitive analysis and by randomizing all data, which can be done efficiently thanks to our hardware support.

Data-Flow Integrity (DFI) [15] and Write Integrity Testing (WIT) [135] also perform alias analysis to build a set of equivalence classes and define a data-flow policy. However, they instrument the code to enforce the data flow and do not randomize the data representation. Among these, DFI is closer to `HARD` as it instruments both the loads and stores, while WIT checks that the data flow is valid only on writes. As `HARD` encrypts before every store and decrypts after every load, it mitigates all attacks that DFI prevents, and as discussed in WIT [135], DFI prevents all attacks that WIT prevents. Thanks to its architectural support, `HARD` also incurs less performance overhead.

HDFI [17] introduced the notion of *Data-flow Isolation*, which allows a program

Table 5.5: This table compares `HARD` *with existing mechanisms in four aspects: (1) The average and worst case performance overhead, (2) If type-safety is required, (3) If compromised external binary have full access to the program memory and (4) If it is prototyped with FPGA. Performance number for Archipelago is omitted as it was not tested with a CPU-intensive benchmark unlike the others.*

	Mechanism Type	**Mechanism**	**(1)**	**(2)**	**(3)**	**(4)**
Software Only	Memory Safety	SoftBound + CETS [16, 145]	116 / 300	Y	Y	N/A
	Safe Allocators	DieHarder [181]	20 / 117	N	N	N/A
		Archipelago [182]	- / -	N	N	N/A
	Data Space Randomization	DSR [162]	14 / 26	N	N	N/A
	Data-flow integrity	DFI [15]	104 / 155	N	N	N/A
Hardware Assisted	Memory Safety	Watchdog [114]	24 / 70	Y	Y	N
		WatchdogLite [115]	29 / 230	Y	Y	N
	Data Space Randomization	`HARD` (This Work)	12 / 42	N	N	Y

to place sensitive data in isolated memory regions effectively and efficiently. This mechanism is similar to `HARD` in that it also relies on hardware support. The HDFI hardware is used to classify instructions and prevent instructions in one group from accessing the memory accessed by instructions in the other group. However, HDFI cannot support more than two such groups because it uses only one bit to define which group a machine word belongs to. In contrast, `HARD` allows the program to classify the memory regions into 2^{11} groups as the underlying hardware uses 11-bit IDs to identify which class an instruction should access. HDFI is not accompanied with an automated way to classify the instructions, instead requiring developers to manually implement an appropriate data-flow isolation policy. `HARD` has a full-fledged DSR mechanism that relieves developers from this burden.

Enforcement of memory safety also mitigates the data-only attacks in general, as most attacks cause the victim program to violate memory safety. Due to the importance of this problem, a number of either software-only or hardware-based mechanisms have been proposed. However, some of these mechanisms cannot handle memory reallocation correctly [183, 184, 141, 185, 186, 187], and others are incompatible with unprotected external code [188, 189, 190, 191]. More recently Softbound [16] used *fat pointers* with disjoint metadata to prevent violation of spatial memory safety and maintain compatibility with unprotected external binaries. Subsequently, CETS [145] was proposed to prevent the violation of temporal safety by using identifier to track the allocation states and disjoint metadata. These mechanisms introduced considerable performance overhead, which was addressed with hardware support in HardBound [113], Watchdog [114] and WatchdogLite [115].

While these mechanisms have shown that enforcement of memory safety can happen at a reasonable performance cost, they require the protected program to be *type-safe* [192]. If not, these mechanisms raise false positive alarms when the program makes type-unsafe casts. HARD, by contrast, does support type-unsafe programs. An additional problem with memory safety-enforcement mechanisms is that they leave external binaries completely unprotected and cannot prevent memory access instructions in external binaries from accessing the data of the hardened program. While DSR does not protect external binaries either, an attacker cannot use a vulnerability in an external library to disclose or predictably corrupt the data of the hardened program because all of the hardened program's data is encrypted.

One type of attack that is mitigated by HARD, as well as safety-enforcement mechanisms, is heap corruption. Specialized memory allocators such as DieHard [193], DieHarder [181] and Archipelago [182] also protect heap objects against corruption. While these allocators have been shown to effectively mitigate attacks on the heap, they also incur non-negligible performance overhead, especially for allocation intensive benchmarks. Recent work has also shown that those allocators are still susceptible to a targeted attack [166].

Table 5.5 summarizes the differences between HARD and closely related work.

5.11 Summary

Although exploit mitigations have improved, those improvements have focused on preventing malicous code execution. Protecting against data-oriented exploits without using expensive memory safety techniques such as bounds checking remains a thorny issue. HARD accelerates data-oriented mitigations in a fully automatic way which is compatible with the vast amounts of existing C and C++ code in use today. We've advanced the state of the art on two axes: performance and security. Security wise, we have increased the odds that data randomization affects malicous flows by using context-sensitive alias analysis to compute equivalence classes. Moreover, we have designed HARD to withstand information leakage attacks against the encryption keys. In terms of performance, our experiments show that hardware-assisted DSR costs less than a third of software-only DSR. Lastly, we have described how we can decrease cost of protection even further.

Chapter 6

Conclusion

This thesis shows a set of hardware-assisted mechanisms that we have devised to mitigate the memory corruption attacks. In addition to the performance benefit that is expected for the dedicated hardware components in general, the hardware techniques could help the mechanisms become more resilient to the attacks on themselves. The hardware components for the attacks on OS kernels could inherently be isolated from them which could be under the control of the attackers. The ones for the DFI enforcement could also protect sensitive memory regions that they use with hardware-based isolation.

Though the idea of introducing dedicated hardware components for OS kernel security is not new, the techniques presented in the first half of this thesis have shown that they could be more efficient and effective. With snoop-based monitoring, the external hardware monitors generated less number of memory access and thus introduced less performance overhead. In addition, they could detect the transient attacks which the monitors analyzing snapshots could not.

Using of the signals from the program trace interface with the snooped bus traffic, we could mitigate a set of attacks on OS kernels more comprehensively. The first step was the complete mitigation of the code-injection attacks. Using the memory access and kernel control-flow information, Kargos could detect any sort of code-injection attacks.

The similar hardware support could be adopted to the detection of a type of code-reuse attacks using the function returns. Once a system adopts these two mechanisms, it would raise the bar significantly, for the attacker who aim to mislead the victim kernel to execute their code.

Architectural supports were also beneficial for the software-oriented mechanisms which can be implemented without the help of the dedicated hardware. Though the addition of such hardware or augmentation of the existing architecture for them could be a burden, we could observe a significant benefit that can motivate such additions or changes.

In HDFI project, the existence of the dedicated hardware support made the software mechanisms be elegant and efficient. The hardware support introduced the performance overhead only when the corresponding software mechanism is active, and the effort of implementing such mechanisms were largely reduced.

HARD project again shows that the architectural supports could have a clear performance and security benefit. The specialized cache and new instructions reduce the performance overhead of HARD by three times on average. In addition, hardware-based isolation made it be resilient against the memory disclosure attacks targeting the keys.

In conclusion, this thesis says that the hardware techniques can both reduce the performance cost of mitigating the memory corruption attacks and help detecting more of them. While this is already a common belief, this thesis provides more evidence that supports the belief by presenting the prototypes and evaluations of the state of the art techniques in several directions of research, on monitoring the OS kernels and mitigating data-flow integrity violations.

Chapter 7

Bibliography

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[2] IDC Research, Inc., “Smartphone OS Market Share, 2015 Q2.” Online, 2015. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

[3] “CVE-2014-3153.” Online, May 2014. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3153`.

[4] “CVE-2015-3636.” Online, May 2015. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3636`.

[5] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, “Sok: Introspections on trust and the semantic gap,” in *IEEE Symposium on Security and Privacy (Oakland)*, May 2014.

[6] “XEN : Vulnerability Statistics.” Online. `https://www.cvedetails.com/vendor/6276/XEN.html`.

[7] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, “Copilot - a coprocessor-based kernel runtime integrity monitor,” in *Usenix Security Symposium (Security)*, 2004.

[8] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, “Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity,” in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[9] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, “Vigilare: Toward snoop-based kernel integrity monitor,” in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[10] 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 Security Symposium (Security)*, 2013.

[11] I. Heo, D. Jang, H. Moon, H. Cho, S. Lee, B. B. K. Kang, and Y. Paek, “Efficient Kernel Integrity Monitor Design for Commodity Mobile Application Processors,” *Journal of Semiconductor Technology and Science*, 2015.

[12] H. Moon, H. Lee, I. Heo, K. Kim, Y. Paek, and B. Kang, “Detecting and preventing kernel rootkit attacks with bus snooping,” *IEEE Transactions on Dependable and Secure Computing*, 2015.

[13] J. Lee, Y. Lee, H. Moon, I. Heo, and Y. Paek, “Extrax: Security extension to extract cache resident information for snoop-based external monitors,” in *Design, Automation & Test in Europe*, 2015.

[14] H. Moon, J. Lee, D. Hwang, S. Jung, J. Seo, and Y. Paek, “Architectural supports to protect os kernels from code-injection attacks,” in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, 2016.

[15] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[16] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Softbound: Highly compatible and complete spatial memory safety for c,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[17] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, “HDFI: Hardware-Assisted Data-Fow Isolation,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[18] C. Song, *Preventing Exploits against Memory Corruption Vulnerabilities*. PhD thesis, Georgia Institute of Technology, Aug. 2016.

[19] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer, “Secure coprocessor-based intrusion detection,” in *Proceedings of the workshop on ACM SIGOPS European workshop*, 2002.

[20] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in *Annual Network and Distributed System Security Symposium (NDSS)*, 2003.

[21] J. Rhee, R. Riley, D. Xu, and X. Jiang, “Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring,” in *International Conference on Availability, Reliability and Security*, 2009.

[22] F. Zhang, J. Wang, K. Sun, and A. Stavrou, “Hypercheck: A hardware-assistedintegrity monitor,” *IEEE Transactions on Dependable and Secure Computing*, 2014.

[23] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," in *In Proceedings of the 7th CanSecWest conference*, 2006.

[24] W. Arbaugh, D. Farber, and J. Smith, "A secure and reliable bootstrap architecture," in *IEEE Symposium on Security and Privacy (Oakland)*, 1997.

[25] B. Kauer, "Oslo: improving the security of trusted computing," in *Usenix Security Symposium (Security)*, 2007.

[26] J. Wei, B. Payne, J. Giffin, and C. Pu, "Soft-timer driven transient kernel control flow attacks and defense," in *Annual Computer Security Applications Conference (ACSAC)*, 2008.

[27] ARM Limited, *AMBA^{TM} Specification*, 1999.

[28] Aeroflex Gaisle, *GRLIB IP Core User's Manual*, January 2012.

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

[30] D. Hellström, *SnapGear Linux for LEON*. Gaisler Research, 2008.

[31] Xilinx, *Zynq-7000 All Programmable SoC Technical Reference Manual*, 2013.

[32] "adore-ng-0.41.tgz." `http://packetstormsecurity.com/files/32843/adore-ng-0.41.tgz.html`. Last accessed Sep 4, 2012.

[33] Cyberwinds, "knark-2.4.3.tgz." `http://packetstormsecurity.com/files/24853/knark-2.4.3.tgz.html`. Last accessed Sep 4, 2012.

[34] Optyx, "Kis 0.9." `http://packetstormsecurity.com/files/25029/kis-0.9.tar.gz.html`. Last accessed Sep 4, 2012.

[35] RaiSe, "Enye lkm rookit modified for ubuntu 8.04." `http://packetstormsecurity.com/files/75184/`

`Enye-LKM-Rookit-Modified-For-Ubuntu-8.04.html`. Last accessed Sep 4, 2012.

[36] `http://packetstormsecurity.com/UNIX/penetration/rootkits`. Last accessed Sep 4, 2012.

[37] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proceedings of international symposium on Recent Advances in Intrusion Detection*, 2008.

[38] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Kernel malware analysis with untampered and temporal views of dynamic kernel memory," in *Proceedings of the international conference on Recent advances in intrusion detection*, 2010.

[39] J. Rhee and D. Xu, "Livedm: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging," tech. rep., 2010.

[40] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[41] R. Love, *Linux Kernel Development*. Addison Wesley, 3 ed., Nov. 2010.

[42] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

[43] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang, "Atra: Address translation redirection attack against hardware-based external monitors," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[44] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms.," in *Usenix Security Symposium (Security)*, 2009.

[45] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert, "Persistent data-only malware: Function hooks without code," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[46] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: attacks, implications and opportunities," in *Proceedings of the Eleventh Workshop on Mobile Computing Systems*, 2010.

[47] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[48] N. A. Quynh and Y. Takefuji, "A novel approach for a file-system integrity monitor tool of xen virtual machine," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.

[49] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[50] K. Kaneda, "Tiny virtual machine monitor." `http://www.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/`.

[51] R. Russell, "Lguest: The simple x86 hypervisor." `http://lguest.ozlabs.org/`.

[52] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[53] D. Hollingworth and T. Redmond, "Enhancing operating system resistance to information warfare," in *Century Military Communications Conference Proceedings*, 2000.

[54] 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*, 2008.

[55] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *IEEE Transactions on Dependable and Secure Computing*, 2011.

[56] 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 (Security)*, 2006.

[57] Y. Bulygin and D. Samyde, "Chipset based approach to detect virtualization malware a.k.a. deepwatch," in *BlackHat USA*, 2008.

[58] D. Clarke, G. E. Suh, B. Gassend, M. van Dijk, and S. Devadas, "Checking the integrity of a memory in a snooping-based symmetric multiprocessor (smp) system," 2004.

[59] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware runtime monitoring for dependable cots-based real-time embedded systems," in *Proceedings of the 2008 Real-Time Systems Symposium*, 2008.

[60] INTEL, *Intel R 64 and IA-32 Architectures Software Developer's Manual Volume 3b: System Programming Guide (Part 2)*, 2013.

[61] ARM, *ARM Architecture Reference Manual, ARM v7-A and V7-R edition, Tech. rep.*, 2012.

[62] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[63] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," in *IEEE Mobile Security Technologies Workshop*, 2014.

[64] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[65] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[66] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou, "Secpod: a framework for virtualization-based security systems," in *ATC Annual Technical Conference (ATC)*, 2015.

[67] ARM, *CoreSight PTM-A9 Technical Reference Manual*, 2011.

[68] Intel, *Intel64 and IA-32 Architectures Software Developer's Manual*, 2014.

[69] 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 *Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[70] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[71] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[72] J. Criswell, N. Dautenhahn, and V. Adve, “Kcofi: Complete control-flow integrity for commodity operating system kernels,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[73] X. Ge, N. Talele, M. Payer, and T. Jaeger, “Fine-grained control-flow integrity for kernel software,” in *2016 IEEE European Symposium on Security and Privacy (EuroSP)*, 2016.

[74] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[75] H. Ozdoganoglu, T. Vijaykumar, C. Brodley, B. Kuperman, and A. Jalote, “Smashguard: A hardware solution to prevent security attacks on the function return address,” *IEEE Transactions on Computers*, 2006.

[76] Samsung, Electronics, *Exynos 4*, 2012.

[77] iVeia, “Building android 4.2.2 bsp on zc702.” `http://www.wiki.xilinx.com/Building+Android+4.2.2+BSP+on+ZC702`, 2015.

[78] L. McVoy and C. Staelin, “Lmbench: Portable tools for performance analysis,” in *ATC Annual Technical Conference (ATC)*, 1996.

[79] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: A new class of code-reuse attack,” in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[80] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity,” in *Usenix Security Symposium (Security)*, 2015.

[81] J. Katcher, “Postmark: A new file system benchmark,” tech. rep., Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

[82] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[83] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing forward-edge control-flow integrity in gcc & llvm,” in *Usenix Security Symposium (Security)*, 2014.

[84] V. van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “Practical context-sensitive cfi,” in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[85] R. Riley, X. Jiang, and D. Xu, “Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing,” in *Recent Advances in Intrusion Detection*, 2008.

[86] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz, “Dynamic hooks: hiding control flow changes within non-control data,” in *Usenix Security Symposium (Security)*, 2014.

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

[88] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, “Sok: Automated software diversity,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[89] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization,” in *Usenix Security Symposium (Security)*, 2012.

[90] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[91] J. J. Gionta, “Prevention and detection of memory compromise.,” 2015.

[92] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “sel4: Formal verification of an os kernel,” in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[93] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein, “seL4 enforces integrity,” in *International Conference on Interactive Theorem Proving*, 2011.

[94] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, “sel4: From general purpose to a proof of information flow enforcement,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[95] T. A. L. Sewell, M. O. Myreen, and G. Klein, “Translation validation for a verified os kernel,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[96] M. Fernandez, G. Klein, and I. Kuz, “Microkernel verification down to assembly,” in *ACM EuroSys Conference*, 2012.

[97] A. V. Fidalgo *et al.*, “Real-time fault injection using enhanced on-chip debug infrastructures,” *Microprocessors and Microsystems*, 2011.

[98] M. Portela-García *et al.*, “On the use of embedded debug features for permanent and transient fault resilience in microprocessors,” *Microprocessors and Microsystems*, 2012.

[99] 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 *Workshop on Hardware and Architectural Support for Security and Privacy*, 2015.

[100] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, “Integration of rop/jop monitoring ips in an arm-based soc,” in *Design, Automation & Test in Europe*, 2016.

[101] M. L. Corliss, E. C. Lewis, and A. Roth, “Using dise to protect return addresses from attack,” *SIGARCH Computter Architecture News*, 2005.

[102] K. Inoue, “Lock and unlock: A data management algorithm for a security-aware cache,” in *IEEE International Conference on Electronics, Circuits and Systems*, 2006.

[103] R. Lee, D. Karig, J. McGregor, and Z. Shi, “Enlisting hardware architecture to thwart malicious code injection,” in *Security in Pervasive Computing*, 2004.

[104] S. Designer, “Getting around non-executable stack (and fix).” `http://seclists.org/bugtraq/1997/Aug/63`, 1997.

[105] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats,” in *Usenix Security Symposium (Security)*, 2005.

[106] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, “Automatic generation of data-oriented exploits,” in *Usenix Security Symposium (Security)*, 2015.

[107] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, “Enforcing Kernel Security Invariants with Data Flow Integrity,” in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[108] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, “Architecture support for defending against buffer overflow attacks,” in *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[109] J. R. Crandall and F. T. Chong, “Minos: Control data attack prevention orthogonal to memory model,” in *International Symposium on Microarchitecture (MICRO)*, 2004.

[110] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, “Branch regulation: Low-overhead protection from code reuse attacks,” in *Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[111] L. Davi, P. Koeberl, and A. R. Sadeghi, “Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation,” in *Annual Design Automation Conference*, 2014.

[112] 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 *Annual Design Automation Conference*, 2015.

[113] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, “Hardbound: Architectural support for spatial safety of the C programming language,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[114] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, “Watchdog: Hardware for safe and secure manual memory management and full memory safety,” in *Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[115] S. Nagarakatte, M. M. Martin, and S. Zdancewic, “Watchdoglite: Hardware-accelerated compiler-based pointer checking,” in *International Symposium on Code Generation and Optimization (CGO)*, 2014.

[116] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, “CHERI: A hybrid capability-system architecture for scalable software compartmentalization,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[117] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer Integrity,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[118] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, “Adapting software fault isolation to contemporary CPU architectures,” in *Usenix Security Symposium (Security)*, 2010.

[119] Oracle, “Introduction to SPARC M7 and application data integrity (ADI).” `https://swisdev.oracle.com/_files/What-Is-ADI.html`, 2015.

[120] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[121] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, “Fast byte-granularity software fault isolation,” in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[122] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[123] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, “ARMlock: Hardware-based fault isolation for ARM,” in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[124] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[125] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A. R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[126] K. J. Biba, "Integrity considerations for secure computer systems," tech. rep., DTIC Document, 1977.

[127] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," tech. rep., DTIC Document, 1973.

[128] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0," Tech. Rep. UCB/EECS-2014-54, UCB, 2014.

[129] Xilinx, "ZC706 evaluation board for the Zynq-7000 XC7Z045 all programmable SoC user guide." `http://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf`, 2015.

[130] Standard Performance Evaluation Corporation, "SPEC CPU2000 benchmark descriptions - CINT 2000." `https://www.spec.org/cpu2000/CINT2000/`, 2003.

[131] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *Annual International Symposium on Computer Architecture (ISCA)*, 2014.

[132] W. S. A. Bradbury and R. Mullins, “Towards general purpose tagged memory.” `http://riscv.org/workshop-jun2015/riscv-tagged-mem-workshop-june2015.pdf`, 2015.

[133] N. Zeldovich, H. Kannan, M. Dalton, , and C. Kozyrakis, “Hardware enforcement of application security policies using tagged memory,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[134] E. Witchel, J. Cates, and K. Asanović, “Mondrian memory protection,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[135] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, “Preventing memory error exploits with WIT,” in *IEEE Symposium on Security and Privacy (Oakland)*, 2008.

[136] J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” in *Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[137] M. Dalton, H. Kannan, and C. Kozyrakis, “Raksha: a flexible information flow architecture for software security,” in *Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[138] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[139] H. Kannan, M. Dalton, and C. Kozyrakis, “Decoupling dynamic information flow tracking with a dedicated coprocessor,” in *International Conference on Dependable Systems and Networks (DSN)*, 2009.

[140] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[141] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *Annual International Symposium on Computer Architecture (ISCA)*, 2008.

[142] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *International Conference on Dependable Systems and Networks (DSN)*, 2012.

[143] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Pump: a programmable unit for metadata processing," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2014.

[144] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[145] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *International Symposium on Memory Management*, 2010.

[146] Intel Corporate, "Intel architecture instruction set extensions programming reference." `https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-refer` 2013.

[147] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Pointguard tm: protecting pointers from buffer overflow vulnerabilities,” in *Usenix Security Symposium (Security)*, 2003.

[148] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, “The rocket chip generator,” tech. rep., EECS Department, University of California, Berkeley, 2016.

[149] UC Berkeley Architecture Research, “Rocket microarchitectural implementation of RISC-V ISA.” `https://github.com/ucb-bar/rocket`, 2015.

[150] A. Ou, A. Waterman, Q. Nguyen, darius bluespec, and P. Dabbelt, “RISC-V Linux Port.” `https://github.com/riscv/riscv-linux`, 2015.

[151] H. Mao, “make sure l2 passes no-alloc acquires through to outer memory.” `https://github.com/ucb-bar/uncore/commit/e53b5072caf12a2c18245cecd709204a4231d2d9`, 2015.

[152] A. Jaleel, “Memory characterization of workloads using instrumentation-driven simulation.” `http://www.glue.umd.edu/~ajaleel/workload/`, 2008.

[153] ARM, “CoreLink^{TM} TrustZone Address Space Controller TZC-380.” `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431c/DDI0431C_tzasc_tzc380_r0p1_trm.pdf`, 2010.

[154] C. Celio, D. A. Patterson, and K. Asanović, “The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor,” Tech. Rep. UCB/EECS-2015-167, UCB, 2015.

[155] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, "Exploiting and protecting dynamic code generation," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[156] PaX Team, "Address space layout randomization." `http://pax.grsecurity.net/docs/aslr.txt`, 2004.

[157] PaX Team, "PaX non-executable pages design & implementation." `http://pax.grsecurity.net/docs/noexec.txt`, 2004.

[158] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, D. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Usenix Security Symposium (Security)*, 1998.

[159] M. Abadi, M. Budiu, Úlfar. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, 2005.

[160] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[161] S. Bhatkar and R. Sekar, "Data space randomization," 2008.

[162] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Tech. Rep. MSR-TR-2008-120, Microsoft Research, 2008.

[163] Xilinx, "ZC702 evaluation board for the Zynq-7000 XC7Z020 all programmable SoC user guide." `http://www.xilinx.com/support/documentation/boards\_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf`, 2015.

[164] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, 2004.

[165] C. Lattner and V. Adve, "Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, 2005.

[166] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[167] T. Rains, M. Miller, and D. Weston, "Exploitation trends: From potential risk to actual risk," in *RSA Conference*, 2015.

[168] Codenomicon and N. Mehta, "The Heartbleed Bug." `http://heartbleed.com/`, 2014.

[169] D. Lea and W. Gloger, "A memory allocator," 1996.

[170] Q. Zeng, M. Zhao, and P. Liu, "Heaptherapy: An efficient end-to-end solution against heap buffer overflows," in *International Conference on Dependable Systems and Networks (DSN)*, 2015.

[171] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, 2014.

[172] D. Jang, Z. Tatlock, and S. Lerner, "SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[173] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, “A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators,” in *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014 - 40th*, 2014.

[174] ARM Ltd., “Cortex-a5 mpcore technical reference manual revision: r0p1.” `https://developer.arm.com/docs/ddi0434/b/1-introduction/11-about-the-cortex-a5-mpcore-processor`. Accessed: 2016-11-14.

[175] M. Schoeberl, “Design and implementation of an efficient stack machine,” in *IEEE International Parallel and Distributed Processing Symposium*, 2005.

[176] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, “Breaking the memory secrecy assumption,” in *Proceedings of the Second European Workshop on System Security*, 2009.

[177] J. Siebert, H. Okhravi, and E. Söderström, “Information leaks without memory disclosures: Remote side channel attacks on diversified code,” in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[178] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, “Undermining information hiding (and what to do about it),” in *Usenix Security Symposium (Security)*, 2016.

[179] C. Lattner, A. Lenharth, and V. Adve, “Making context-sensitive points-to analysis with heap cloning practical for the real world,” *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[180] B. Steensgaard, “Points-to analysis in almost linear time,” in *ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[181] G. Novark and E. D. Berger, “Dieharder: Securing the heap,” in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[182] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, “Archipelago: Trading address space for reliability and security,” in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[183] D. Dhurjati and V. Adve, “Backwards-compatible array bounds checking for c with very low overhead,” in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[184] R. W. M. Jones and P. H. J. Kelly, “Backwards-compatible bounds checking for arrays and pointers in c programs,” in *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, 1997.

[185] S. Ghose, L. Gilgeous, P. Dudnik, A. Aggarwal, and C. Waxman, “Architectural support for low overhead detection of memory violations,” in *Design, Automation Test in Europe Conference Exhibition*, 2009.

[186] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Memtracker: Efficient and programmable support for memory access monitoring and debugging,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[187] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, “Memtracker: An accelerator for memory debugging and monitoring,” *ACM Transactions on Architecture and Code Optimization (TACO)*, 2009.

[188] T. M. Austin, S. E. Breach, and G. S. Sohi, “Efficient detection of all pointer and array access errors,” in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994.

[189] H. Patil and C. Fischer, “Low-cost, concurrent checking of pointer and array accesses in c programs,” *Softw. Pract. Exper.*, 1997.

[190] W. Xu, D. C. DuVarney, and R. Sekar, “An efficient and backwards-compatible transformation to ensure memory safety of c programs,” in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT ’04/FSE-12, (New York, NY, USA), pp. 117–126, ACM, 2004.

[191] W. Chuang, S. Narayanasamy, and B. Calder, “Accelerating meta data checks for software correctness and security,” *Journal of InstructionLevel Parallelism*, 2007.

[192] M. Hicks, “What is memory safety?.” `http://www.pl-enthusiast.net/2014/07/21/memory-safety/`, 2014.

[193] E. D. Berger and B. G. Zorn, “DieHard: probabilistic memory safety for unsafe languages,” in *ACM SIGPLAN Notices*, vol. 41, pp. 158–168, ACM, 2006.

초 록

많은 컴퓨터 프로그램들이 성능 등의 이유로 C나 C++ 와 같은 안전하지 않은 언어로 작성되어 있으며, 또한 매우 많은 코드로 매우 복잡하게 구현되어 있다. 이와같은 특성들은 프로그램들이 공격의 수단이 되는 취약점을 가질 수 밖에 없게 만든다. 이런 취약점들이용해 공격자들은 프로그램의 메모리에 임의로 접근할 수 있으며, 이를 통해 최종적으로 다양한 목적을 달성할 수 있다. 이와 같은 문제점에도 불구하고, 프로그램들로부터 모든 취약점을 제거하는 것은 일반적으로 불가능하다고 알려져있다. 비록 일부 프로그램에 대해서는 특정 형태의 취약점을 갖지 않는다는 증명이 가능하지만, 그러한 증명이 가능한 것은 비교적 간단한 프로그램 뿐이며, 모든 종류의 취약점이 없다는 것을 증명할 수 있는 것 또한 아니다. 때문에 이렇게 취약점들을 이용하는 공격에 대응하기 위한 많은 기법들이 제안되어왔다.

이 논문은 앞에서 언급한 취약점을 이용하는 메모리 변조 공격들에 대응하기 위한 하드웨어 기반의 기법을을 제시하고 있다.

이 논문의 전반부에서는 운영체제 커널에 대한 공격들에 대응하는 기법들을 다룬다. 많은 컴퓨터 시스템에서 운영체제 커널은 그 시스템에 대한 모든 권한을 가지고 있으며, 이를 이용해 모든 자원에 접근할 수 있다. 어떤 프로그램이든 파일, 네트워크, 심지어 메모리나 프로세서 등의 자원에 접근하기위해서는 운영체제 커널을 호출해야 한다. 이와 같은 권한과 특성을 가진 커널이 공격자에 의해 변조되어 공격자가 원하는 형태로 모든 연산을 처리할 경우, 공격자는 결과적으로 그 시스템에서 동작하는 모든 프로그램에 임의의 영향을 줄 수 있는데, 이러한 점은 커널을 어느 시스템을 공격하고자 하는 공격자에게 있어 매우 매력적인 공격 대상이 되도록

한다.

커널에 대한 공격에 대응하는 기법은 바로 그 커널이 시스템에 대한 모든 권한을 갖고 있다는 점을 반드시 고려하여 설계되어야 한다. 어떠한 기법이라도 커널에 대한 모든 공격을 무력화할 수 없기 때문에, 별도의 수단을 이용하지 않는 경우 어떠한 기법이라도 커널에 의해 무력화 될 수 있다. 이는 다시말하면 커널이 어떤 한가지 경로로 공격자의 제어하에 들어간 경우, 커널에 의존하는 어떠한 기법도 공격자에 의해 무력화될 수 있다는 것이다. 이 때문에 많은 대응기법들이 운영제체 커널에 의존하지 않는 시스템 구성요소를 이용해 운영체제 커널로부터의 독립성을 확보하고자 하였다. 이 논문에서는 이와 같은 연구들 중 새로운 하드웨어를 추가하고 이에 의존하는 최신 기법을 제시하고 있다. 우리는 연구 과정에서 물리적으로 운영체제 커널로 부터 분리되어 동작하는 하드웨어가 커널의 수행 과정에서 발생하는 여러 정보를 수집하는 효과적이고 효율적인 방법을 설계 및 구현하였으며, 이를 이용하여 운영체제 커널에 대해 알려진 여러 공격 유형에 대해 대응하는 기법을 제시하였다.

이 논문의 후반부에서는 보다 일반적인 메모리 변조공격에 대응할 수 있는 하드웨어 기반의 기법들을 제시한다. 메모리 변조 공격의 중요성으로 인해 그에 대응하기 위한 다양한 기법들이 제시되어왔지만, 대부분의 경우 실용화되기는 어려운 부분을 가지고 있었다. 몇몇 기법들은 하위 호환성의 문제로 인해 위양성 을 방지하기 위한 프로그램 수정을 필요로 했으며, 대부분의 경우 실용화되기에는 큰 성능부하를 가지고 있었다.

이 논문에서는 메모리 변조 공격에 대한 실용적인 대응 기법을 설계하고자 하였으며, 그 결과로 두 가지 기법들을 제시하고있다. 그 중 첫 번째 기법은 프로그램들로 하여금 민감한 데이터의 흐름을 그렇지 않은 데이터의 흐름과 분리할 수 있도록 해준다. 이를 적용한 경우, 공격자는 민감한 데이터를 변조하기 위해 반드시 민감한 데이터에 접근하도록 허용된 코드의 취약점만을 이용해야 하며, 이는 공격자가 활용할 수 있는 취약점의 출현 가능성을 줄임으로써 그들이 민감한 데이터에 접근하는 것을 보다 어렵게 만든다. 두 번째 기법은 공격자들이 공격을 안정적으로 수행하는 것을 방해한다. 이 기법이 적용된 프로그램은 데이터를 랜덤화 하는 방법으로, 프로그램의 메모리 명령어들이 정적 분석을 통해 접근이 허용된 메모리상의 객체에

접근하는 경우에만 정상적으로 동작하고, 허용되지 않은 객체에 접근하는 경우 예측할 수 없는 행동을 하게 만든다. 일반적으로 프로그램에 대한 메모리 변조 공격은 메모리 명령어들이 앞에서 언급한 정적 분석의 결과를 따르지 않는 메모리 접근을 하게 만들기 때문에, 제시된 기법이 적용된 경우 공격자가 의도한 메모리 접근들은 예측할 수 없는 형태로 동작하여 결과적으로 공격이 실패하도록 만들 것이다.

요약하면, 이 논문은 운영체제 커널 또는 일반적인 프로그램들에 적용가능한 하드웨어 기반의 메모리 변조 공격 대응 기법을 네 가지 소개하고 있다.

주요어: 메모리 변조 공격, 운영체제 커널, 코드 삽입 공격, 코드 재사용 공격, 데이터 흐름 무결성, 스눕 기반 모니터링

학번: 2010-20795