CPU TRANSPARENT PROTECTION OF OS KERNEL AND HYPERVISOR INTEGRITY WITH PROGRAMMABLE DRAM

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Ziyi Liu

August 2013

CPU TRANSPARENT PROTECTION OF OS KERNEL AND HYPERVISOR INTEGRITY WITH PROGRAMMABLE DRAM

Ziyi Liu

APPROVED:

Weidong Shi, Committee Chair
Dept. of Computer Science

Shou-Hsuan Huang
Dept. of Computer Science

Zhiqiang Lin
Dept. of Computer Science
University of Texas at Dallas

Dean, College of Natural Sciences and Mathematics

Acknowledgements

This thesis would not be possible without the support of a large number of people who have helped me during my three years at Houston.

In particular, I would like to thank my advisor, Weidong Shi, for his support, patience, and especially his trust and respect. He has shown me how to organize and present ideas, and how to find important and meaningful topics for research.

The inspiration for this work certainly stems from Professor Weidong Shi, and Professor Zhiqiang Lin. Their knowledge in computer architecture and security area enlightened me so much such that I could bypass obstacles and finish the paper. Professor Weidong Shi, Professor Zhiqiang Lin, and Professor Shou-Hsuan Huang served on my thesis committee. I am indebted to them for their efforts to make this thesis better.

Another person who made this possible, perhaps without truly understanding his contribution, has been Yuanfeng Wen. Yuanfeng has been an invaluable sounding board for ideas, a welcoming ear to occasional frustrations, critical eye for my writing, and more importantly a great friend. Special thanks to Tao Feng, Dr. Jonghyuk Lee, and Dr. Dong Hyuk Woo for their tremendous help and encouragement throughout the past three years.

Finally, I am truly grateful to my family for their love, understanding, and encouragement.

CPU TRANSPARENT PROTECTION OF OS KERNEL AND HYPERVISOR INTEGRITY WITH PROGRAMMABLE DRAM

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Ziyi Liu

August 2013

Abstract

Increasingly, cyber attacks (e.g., kernel rootkits) target the inner rings of a computer system, and they have seriously undermined the integrity of the entire computer system. To eliminate these threats, it is imperative to develop innovative solutions running below the attack surface. This paper presents MGuard, a new most inner ring solution for inspecting the system integrity that is directly integrated with the DRAM DIMM devices. More specifically, we design a programmable guard that is integrated with the advanced memory buffer of FB-DIMM to continuously monitor all the memory traffic and detect the system integrity violations. Unlike the existing approaches that are either snapshot-based or lack compatibility and flexibility, MGuard *continuously* monitors the integrity of all the outer rings including both OS kernel and hypervisor of interest, with a greater extendibility enabled by a programmable interface. It offers a hardware drop-in solution transparent to the host CPU and memory controller. Moreover, MGuard is isolated from the host software and hardware, leading to strong security for remote attackers. Our simulation-based experimental results show that MGuard introduces no speed overhead, and is able to detect nearly all the OS-kernel and hypervisor control data related rootkits we tested.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivation

To gain more control over the system and make the attack stealthy, increasingly cyber attacks target the inner rings from OS kernel, virtualization, to even the hardware. Kernel rootkits (e.g., [7]), virtualization rootkits (e.g., Blue Pill [36] and SubVert [26]), and PCI rootkits [18] all represent such a trend, and they all tend to compromise the inner rings (e.g., OS kernels, virtualizations, and hardware interfaces) of a computer system, as well as stealthily facilitate and conceal other add-on attacks. Therefore, these inner ring threats seriously undermine the integrity of the entire computer system and make many defensive solutions running at the same privilege level ineffective.

As these inner ring threats seriously undermine the integrity of the entire computer system, numerous techniques have been proposed to defend against these low level attacks, such as using a specification or data structure guided approach (e.g., [33, 9, 21]), signatures (e.g., using structure field invariants [3, 10], or graph-invariants [27]). While these techniques are certainly promising, the practical issue is where we should deploy the security mechanism. Obviously, we should not deploy them above or within the same ring; otherwise they will be directly tampered by the rootkits running with the same privilege level.

Creating a security software to discover and remove malicious software is fundamentally hard, as both the security and malicious software share the same resources and devices. For instance, tools designed at the application ring to gather volatile memory data can print unreliable data when a rootkit does kernel level modification, and tools designed at the kernel ring such as Linux page fault handler can be hijacked to present a false valid view of the kernel and hide the kernel level rootkit from being detected by software [7].

With the recent advances in virtualization, more solutions have been pushed down to the inner rings. In particular, since hypervisor controls the outer computer systems, a number of techniques use virtual machine introspection (VMI) [16] to detect the kernel rootkits (e.g., [24, 14, 15]). Their assumption is that hypervisor is secure and can be trusted (e.g., [16]). Unfortunately, like any other software layer, a hypervisor can have vulnerabilities and is prone to attacks or unexpected failures.

For instance, in the past few years, we have witnessed a number of successful hypervisor subversions, such as Bluepill [36], SubVert [26] and SubXen [44].

Since the hypervisior approach is still vulnerable, hardware-assisted approaches have been naturally proposed. More specifically, a number of recent studies have explored the possibility of leveraging the existing x86 features, namely system management mode (SMM), to acquire the necessary memory contents for monitoring the hypervisor (e.g., HyperSentry [2] and HyperCheck [39]) without any extra hardware. Unfortunately, the security of SMM cannot be taken for granted [11]. It was demonstrated that SMM handler can be tampered and modified by SMM rootkits [13].

It is thus imperative to design *flexible* and *drop-in* hardware-based solutions compatible with the exiting computer platforms for monitoring the kernel integrity. Copilot [32] is such an example and it uses a dedicated PCI device to monitor the kernel memory integrity. Unfortunately, there are attacks [35] that can prevent PCI-based RAM acquisition devices from correctly accessing the physical memory. This is because PCI devices are located far from the CPU and physical memory, thus requests for accessing the physical memory have to go through multiple hardware components situating in the I/O controller hub, memory controller hub, or the CPU with integrated memory controller hub. These components can be modified and configured by rootkits such that the PCI device could be presented with a different view of the physical memory than what is seen by the CPU, and as such those tampered physical memory areas will be hidden from the analysis [35].

Moreover, similar to HyperSentry and HyperCheck, Copilot uses a memory snapshot-based approach and it cannot detect the transient attacks [43] that happen in between the snapshots. Recently, Vigilare [31] also recognized the issues in snapshot-based approach and demonstrated a low speed bus snooping technique that snoops the

bus traffic between an embedded processor and memory controller. Furthermore, all these existing hardware-assisted approaches including Vigilare only support limited security policies, and cannot perform other defensive functions such as runtime response.

In addition, a number of recent studies have explored the possibility of leveraging existing x86 features, namely system management mode (SMM), to acquire the necessary memory contents for monitoring the hypervisor (e.g., HyperSentry [2] and HyperCheck [39]). SMM is a feature that was introduced on the Intel 386SL and allows an OS to be interrupted and normal execution to be temporarily suspended to execute SMM code at a very high privilege level. It is normally configured at boot-time by the BIOS and the OS has zero knowledge about it. Unfortunately, recent studies showed that the security of SMM cannot be taken for granted [11], and it was demonstrated that SMM handler can be tampered and modified by SMM rootkits [42, 13].

As such, we have to develop the solutions from the most inner ring (i.e., the hardware ring). While there are a number of early attempts from hardware perspective, current solutions are limited in supporting advanced monitoring, detection, and response mechanism. In particular, nearly all the hardware-assisted approaches including PCI-based Copilot [32], SMM-assisted HyperCheck [39] and HyperSentry [2] are kernel memory snapshot-based integrity checkers. Obviously, snapshot-based approaches cannot detect the transient attacks that happen in between the snapshots. Moreover, they only support limited security policies, and cannot perform other defensive functions such as runtime response.

1.2 Research Approach and Objectives

In this thesis we present a new approach from the most inner ring, to enable *programmable*, *high speed*, and *continuous* monitoring and response of system integrity of interest. Similar to the network firewalls that inspect all the traffic pass-through, our system is called MGuard, and it is a memory firewall that monitors the communication between the CPU and the physical memory. As such, compared with all the other systems, the distinctive feature of MGuard is that it can *continuously* monitor all the abnormal memory data passing through with a very *high speed*. Moreover, unlike the PCI-based approaches, MGuard is integrated with the physical memory itself by extending the advanced memory buffer (AMB), and there is no other attack surface other than physical attacks by design.

Our objective is to design an off-CPU and stand-alone solution for monitoring physical memory states of interested kernel spaces, which is as close to the physical memory itself as possible (preventing tampering from the chipset rootkits) and compatible with the existing computer hardware (requiring no change from the existing micro-processors and chipsets). To realize MGuard, we are facing a number of new challenges. Since it is common that modern commercial processors have integrated memory controller hub, it is critical that the solution should be transparent to and compatible with the existing systems. We use fully buffered DRAM as our design target because it is designed in a point-to-point favor which is an adopted topology in the next generation DDR4 standard. While in this paper we demonstrate our technique by extending the AMB of fully buffered DIMM (FB-DIMM) [23, 22], the

principle and concept behind can actually be applied to other DRAM techniques and standards. For example, our solution can be used to the emerging DDR4 by integrating our new components with the DDR4 switch fabric, a topic of future research. For fully buffered DRAM, it is crucial that we have to make sure that there is no performance penalty with the inspection of the memory traffic because such overhead will likely violate DRAM time constraint and render the solution useless.

1.3 Research Contributions of this Thesis

To this end, we have designed a programmable hardware guard that is integrated with the AMB of FB-DIMM to continuously monitor all the memory traffic and detect kernel integrity violations. Because of such design, MGuard provides a programmable interface and allows customized security policies to check the integrity and invariant violation of all the outer rings including both OS kernel and hypervisor of interest. On top of MGuard, we design a number of system integrity checkers, and our experimental results show that MGuard introduces no performance overhead (because it does not introduce any additional latency on the critical path), and is able to detect all the outer ring control data related rootkits including SMM, hypervisor and OS-kernel we tested. In short, this paper makes the following contributions:

- We present a new off-CPU and stand-alone solution, MGuard, to check the system integrity of interest. Unlike the existing solutions that provide limited capability, our system is fully *programmable* and allows both detection and response;

- We have implemented MGuard by leveraging the existing open source IP blocks. We evaluate the MGuard performance based on cycle-based architectural and FB DRAM simulators;
- On top of MGuard, we design a number of kernel control data integrity checkers, which continuously check the memory traffic to system call table (SCT), interrupt handler table (IDT), SMM handler, etc.; and
- Our empirical evaluation results show that MGuard has less than extra 3.5% power consumption and almost no performance overhead. It can detect a wide range of tested rootkits include 11 kernel rootkits, 3 hypervisior rootkits, and 3 SMM rootkits.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 gives background information, including a brief introduction to recently cyber vulnerabilities, and a review of previous work on security platforms on preventing cyber attacks.

Chapter 3 depicts the overview of the system we developed in this thesis. First, a general description of the threat modeling is proposed. Then we present a extended hardware advanced memory buffer that are used to capture and analysis memory traffic. We show how MGuard boots up and how it works.

The design and schematics of our system is presented in Chapter 4. We describe four main function of MGuard in this section: frame interception, address mapping,

active probing, and kernel state monitoring. The structure and function of logics are discussed. We implements the system in both RTL design and full system simulation. We give a simple example how MGuard protects the integrity of kernel.

In order to prove MGuard is working, Chapter 5 analysis the system from security perspective. The system degradation information and its analysis are shown in Chapter 6.

Chapter 7 summarizes the thesis conclusion and provides directions for future work.

A short manual of how to hookup GEM5 and DRAMSIM2 is given in Appendix A.

Chapter 2

Background

This chapter introduces background information and reviews previous work in the relevant areas. Section 2.1 describes what is rootkits and how rootkits generally work. Section 2.2 gives a review of previous work on how system and hardware diversity could detect the rootkits and prevent malicious code execution. The protection approaches include virtualization-based detection, hardware-assist detection, and extra-hardware detection.

2.1 Cyber Attacks and Rootkits

A rootkit is a type of malicious software that is activated each time your system boots up. Rootkits are difficult to detect because they are activated before your system's Operating System has completely booted up. A rootkit often allows the installation of hidden files, processes, hidden user accounts, and more in the systems

OS. Rootkits are able to intercept data from terminals, network connections, and the keyboard.

Rootkits have two primary functions: remote command/control (back door) and software eavesdropping. Rootkits allow someone, legitimate or otherwise, to administratively control a computer. This means executing files, accessing logs, monitoring user activity, and even changing the computer's configuration. Therefore, in the strictest sense, even versions of VNC are rootkits. This surprises most people, as they consider rootkits to be solely malware, but in of themselves they are not malicious at all. One famous (or infamous, depending on your viewpoint) example of rootkit use was Sony BMG's attempt to prevent copyright violations. Sony BMG did not tell anyone that it placed DRM software on home computers when certain CDs were played. On a scary note, the rootkit hiding technique Sony used was so good not one antivirus or anti-spyware application detected it.

There are several types of rootkits, but we will start with the simplest one. User-mode rootkits run on a computer with administrative privileges. This allows user-mode rootkits to alter security and hide processes, files, system drivers, network ports, and even system services. User-mode rootkits remain installed on the infected computer by copying required files to the computers hard drive, automatically launching with every system boot. Sadly, user-mode rootkits are the only type that antivirus or anti-spyware applications even have a chance of detecting. One example of a user-mode rootkit is Hacker Defender. It's an old rootkit, but it has an illustrious history. If you read the link about Hacker Defender, you will learn about Mark Russinovich, his rootkit detection tool called Rootkit Revealer, and his

cat-and-mouse struggle with the developer of Hacker Defender.

Malware developers are a savvy bunch. Realizing that rootkits running in user-mode can be found by rootkit detection software running in kernel-mode, they developed kernel-mode rootkits, placing the rootkit on the same level as the operating system and rootkit detection software. Simply put, the OS can no longer be trusted. One kernel-mode rootkit that's getting lots of attention is the Da IOS rootkit, developed by Sebastian Muniz and aimed at Cisco's IOS operating system. Instability is the one downfall of a kernel-mode rootkit. If you notice that your computer is blue-screening for other than the normal reasons, it just might be a kernel-mode rootkit. Rootkit developers, wanting the best of both worlds, developed a hybrid rootkit that combines user-mode characteristics (easy to use and stable) with kernel-mode characteristics (stealthy). The hybrid approach is very successful and the most popular rootkit at this time.

Firmware rootkits are the next step in sophistication. This type of rootkit can be any of the other types with an added twist; the rootkit can hide in firmware when the computer is shut down. Restart the computer, and the rootkit reinstalls itself. The altered firmware could be anything from microprocessor code to PCI expansion card firmware. Even if a removal program finds and eliminates the firmware rootkit, the next time the computer starts, the firmware rootkit is right back in business.

Virtual rootkits are a fairly new and innovative approach. The virtual rootkit acts like a software implementation of hardware sets in a manner similar to that used by VMware. This technology has elicited a great deal of apprehension, as virtual rootkits are almost invisible. The Blue Pill is one example of this type of rootkit.

2.2 Related Work

Virtualization-based detection As hypervisor is positioned underneath the ring of OS, it can be naturally used for inspecting the OS kernel integrity. Livewire [16] pioneered virtual machine introspection (VMI) and numerous efforts have been focused on how to extend VMI to detect such as kernel rootkits. Notable examples include such as VMwatcher [24], Lycoisid [25], and they all infer kernel rootkit presence using a cross-view comparison approach. HookSafe [41] intercepts guest-kernel function calls to check the integrity; KOP [9] and OSck [21] collects all kernel function pointers through source code analysis and traverses the kernel memory for this purpose.

However, as alluded in Chapter 1, today's hypervisors often have a large code base (with hundreds of thousands lines of code), and it is challenging to have a bug-free implementation. For instance, there are still hundreds of vulnerabilities being found recently in popular hypervisors such as Xen and VMware ESX, as summarized in HyperWall [37]. While HyperSafe [40] enforces a lightweight control flow integrity to protect the hypervisor from being compromised, it is still a software-only approach and requires rewrite the hypervisor.

Hardware-assisted detection Through leveraging system management mode (SMM) present in modern hardware, HyperSentry [2] monitors the integrity of hypervisors using an agent planted in the SMM. Similarly, HyperCheck [39] also leveraged the SMM feature but with the cooperation from a PCI device. Unfortunately, SMM is not secure and can be attacked by SMM rootkits [13]. Flicker [30] provides a

framework to isolate sensitive code execution and attestation, by using the new processor features in modern x86 CPU. However, Flicker requires the cooperation from OS and applications. HyperWall [37] extends the instruction set and protects guest VMs from a compromised hypervisor. It also requires the guest VM cooperations. While our MGuard is also a hardware-assisted approach, it works transparently to the legacy systems.

Extra-hardware-based detection For convenience or transparency, extra hardware was created for acquiring the contents of system memory without OS or CPU interaction (e.g., [32]). In particular, Copilot provides a PCI-based solution for checking the integrity of system memory by issuing PCI DMA requests periodically to take snapshot of the physical memory of a live system. Unfortunately, PCI-based approach can be bypassed and rootkits can modify the PCI configurations and bridge settings.

Most recently, Vigilare [31] leverages bus snooping techniques with an extra hardware to detect the system integrity, and it is able to capture the transient manipulations of kernel memory. While at high level, both MGuard and Vigilare recognized the problem of the transient attacks, the solutions and targeted environments are actually very different. The major concern about their technique as acknowledged by the authors is: Vigilare has host bus bandwidth limitations. The bandwidth of the high-end modern server may exceed the computing speed of a slow embedded processor which is used in their experiment. In this case, Vigilare is not suitable for high-end DRAM system running at much higher speed based on point-to-point links such as FB-DIMM. Another significant difference between Vigilare and our solution

is that Vigilare snoops the bus between an embedded processor core and the memory controller, which makes Vigilare unpractical as a real solution not only because specific snoopers are required for different processors, but also because today's commercial processors have integrated memory controller and the bus between processor cores and memory controller is hidden and inaccessible by a snoop device. Such problems don't exist in MGuard because MGuard is integrated with DRAM DIMM devices and is transparent to the memory controller and host CPU. In addition, with the DRAM page lookup table for filtering out uninterested traffic and tracking updates to interested kernel space automatically, MGuard is much more efficient than Vigilare.

By extending chipset-specific uController and internal DMA, DeepWatch [8] watches system memory via DMA and scan for signatures of known VT-x based hypervisor rootkits such as malicious `vmexit` handler and SMM rootkits. However, like many other approaches, DeepWatch is also snapshot-based, and it cannot detect the transient attacks. In addition, there are also TPM chipset to enable trusted computing in commodity hardware. However, TPM approaches typically do not support the continuous monitoring of system integrity, and they are either used to ensure the trusted booting or sealed storage (c.f., [30]).

The System	Software-only approach	Hardware-assisted approach	Extra-hardware approach	CPU transparent&drop-in solution	Protecting OS kernel	Protecting Hypervisor	No cooperation from OS	No cooperation from Hypervisor	No source code access	Snapshot-based monitoring	Continuous monitoring	Flexible security policies
VMwatcher	✓				✓		✓			✓		✓
Lycosid	✓				✓		✓		✓	✓		✓
KOP	✓				✓					✓		✓
HookSafe	✓				✓		✓		✓		✓	
HyperSafe	✓					✓	✓				✓	
HyperSentry		✓				✓	✓	✓	✓	✓		✓
HyperCheck		✓	✓			✓	✓	✓	✓	✓		✓
Flicker		✓			✓	✓					✓	✓
HyperWall		✓				✓	✓				✓	✓
MGuard		✓		✓	✓	✓	✓	✓	✓		✓	✓
Vigilare			✓		✓	✓	✓	✓	✓		✓	✓
Copilot			✓	✓	✓	✓	✓	✓	✓	✓		✓
DeepWatch			✓	✓		✓	✓	✓	✓	✓		✓

Table 2.1: Summary of the Related Work Comparison.

Chapter 3

System Overview

This chapter presents the overview architecture we have developed. We introduce the whole system start from its threat model, background of FB-DIMM, goals and challenges and overall design. Section 3.1 describes the threat model of the problem. Section 3.2 presents the general approach of analyze intrusions and rootkits by using memory dumping technique. Section 3.3 discusses the structure of FB-DIMM and Advanced Memory Buffer(AMB). Section 3.4 introduces the design objectives and requirements as well as the challenges when design MGuard. Finally, Section 3.5 discusses the overall solution.

3.1 Threat Model

As an extended hardware in AMB, MGuard is effective for attackers who have gained the administrator's privilege on the host system through such as remote exploits.

Particularly, when the OS has already been compromised by the attackers, MGuard is able to find out suspicious modifications. In addition, the attackers are unable to access MGuard, because the design is transparent to both OS and users. However, mitigating physical attacks by an insider who has direct full control to machine hardware is not the objective of our scheme.

System rootkits including kernel and hypervisor rootkits, which run with the most inner rings, are the primary threats prevented by MGuard. Detecting system rootkits is one of the grand challenges because software only approaches often fail for reasons mentioned in Chapter 1. We assume that all the software layer has been compromised by attackers, and attackers leave such as backdoors by modifying the kernel and hypervisor. For example, the attackers can install kernel and hypervisor rootkits that place hooks on critical system calls. The details of threat detection will be covered in Chapter 5.

3.2 Kernel Inspection from Physical RAM Image

To analyze intrusions and rootkits, it is a common practice to dump physical RAM image from a live system and extract critical kernel information from the dumped physical image. To this end, comprehensive set of software tools were developed to examine dumped physical RAM contents. Special hardware can be used to take a snapshot of physical DRAM to ensure a truthful kernel image is obtained. For continuous and real-time kernel inspection, previous schemes have experimented with PCI-based [32] and bus snoop-based approaches [31]. Our approach is one step

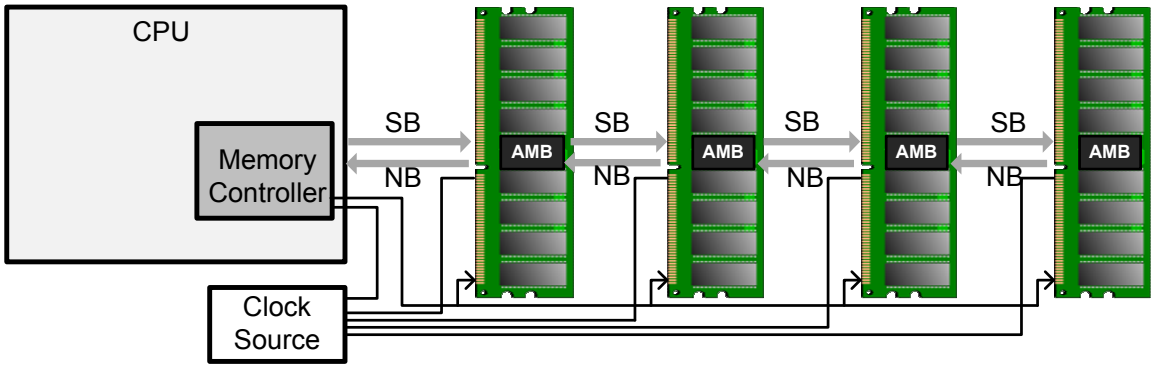

Figure 3.1: Architecture Overview of FB-DIMM

further that integrates such capabilities directly into the DRAM DIMM devices for a drop-in solution with no performance overhead and transparent to the host hardware.

3.3 Background on FB-DIMM

In modern computer systems, higher memory bandwidth is often required to meet the need of higher CPU speeds. As a memory technology that can be used to increase scalability, reliability, and density of the memory systems, FB-DIMM [23, 22] was designed for the server market. For instance, Intel recently has adopted FB-DIMM technology for their Xeon 5000/5100 series and beyond. Rather than using the traditional memory devices that communicate through parallel bus (e.g., multi-drop buses), the DRAM devices are buffered behind one or more Advanced Memory Buffer (AMB) devices. The system memory controller connects to the AMB using high speed serial communication links.

As shown in Fig. 3.1, the memory controller in CPU communicates with the

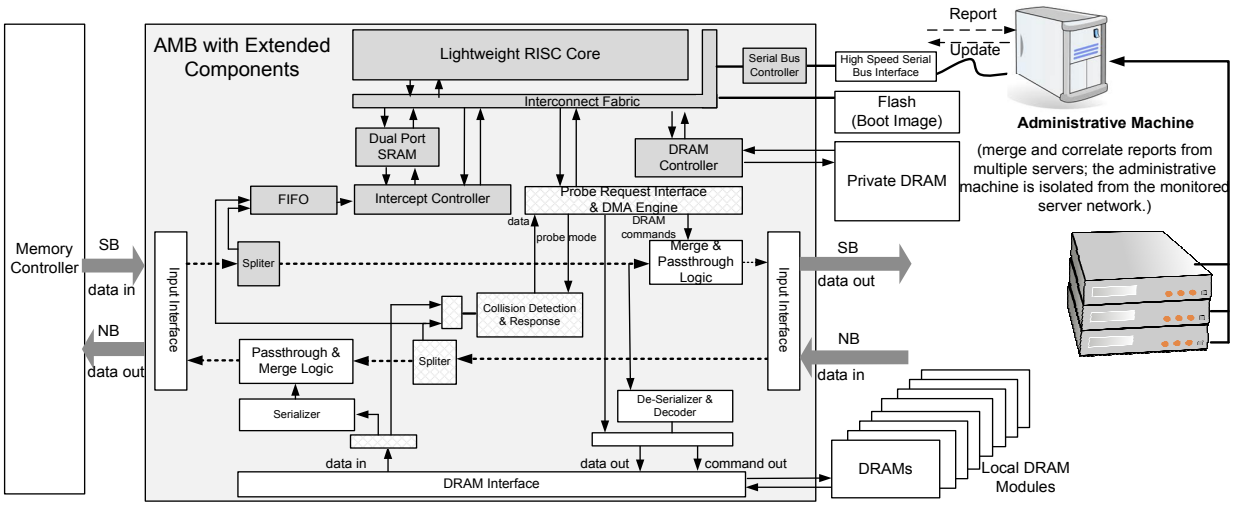

Figure 3.2: AMB with Extended Components

AMBs through serial interface. The daisy-chained topology provides an extension from a single DIMM per channel to up to 8 DIMMs per channel. The channel interconnection actually consists of two unidirectional links: one in the southbound direction and the other in the northbound direction. The memory controller sends data to the first DIMM via southbound link. Whenever the DIMM receives the data, it will forward the data to the next DIMM until the last DIMM receives the data. Similarly, the DIMM sends back the data to the next DIMM via northbound link until memory controller is reached. The AMBs can be considered as dumb forwarding devices.

3.4 Goals and Challenges

The objective of MGuard is to design a new off-CPU technique for capturing memory states with the capability of, (i) obtaining volatile memory states in a reliable

and timely manner using solutions as closer to the physical RAM as possible, (ii) providing a stand-alone and drop-in solution that is compatible with the existing hardware platforms and standards without modifying the processor or chipsets, and (iii) capturing volatile memory states without causing any performance overhead of the entire computer system.

Our solution is to leverage the potentials provided by the FB-DIMM to integrate new hardware components into the AMB chip so that the extended AMB can provide secure, reliable, and timely capture of the physical memory states. Because AMB is the closest logic device to the DRAM modules, our solution can ensure truthful views of the physical memory. Furthermore, our approach offers a drop-in solution to the existing FB-DIMM-based platform because it does not require any changes of the existing chipsets or processors that are already deployed. More importantly, our approach has the advantage that it does not require a modified memory controller because it is unrealistic to change the commercial processors that already have integrated with memory controllers.

However, many great challenges must be addressed in order to achieve the above aggressive goals. In particular, in FB-DIMM architecture, AMB is treated as a passive synchronous device. It acts like a pass-through switch, directly forwarding the requests that it receives from the memory controller to successive DIMMs and forwarding frames returned from southern DIMMs to the memory controller. Frame scheduling is performed exclusively by the memory controller. The AMB only converts the serial protocol to DDRx based commands without implementing any scheduling functionality. This means that the AMB has no slack time or freedom

to add extra processing steps necessary for logging memory states, otherwise it will violate the timing as specified by the memory controller.

3.5 MGuard Overview

Our solution to these challenges is to move all the new components off the critical paths of AMB, in contrast to Vigilare [31] that snoops the bus and is bounded with the bus speed. Consequently, our solution is as close to the physical memory as one can possibly get by integrating programmable detection modules directly with the DRAM DIMM devices. Our solution is compatible with the FB-DIMM standards [23, 22] and fully programmable by using a general purpose RISC core. It incurs virtually no latency overhead because it does not introduce any additional latency on the critical path; otherwise the overhead would likely violate DRAM command timing enforced by the host memory controller.

A high level view of the design is presented in Fig. 3.2. We extend the AMB with the following new components: a programmable RISC core, a DRAM controller connecting to a private DRAM (hidden from the host CPU but accessible by the programmable RISC core), a set of components that capture incoming/outgoing DRAM frames and preserve interested DRAM pages (e.g., those storing kernel data and codes) to the private DRAM, and a set of components for issuing DRAM read commands into the southbound frame stream under control of the RISC core. The extended AMB supports two modes of operations: intercept mode and probing mode. More specifically:

- In intercept mode, both NB and SB DRAM frames are captured and analyzed according to the policies stored in a dual-ported SRAM. If they fall into the ranges of interested DRAM pages, the DRAM data contained in the DRAM frames will be copied and saved to the private DRAM for later analysis by the RISC core. Both write update from the host and data in response to the read request from the host are intercepted. In the intercept mode, the private DRAM will provide the most recent view of the important kernel memory states. We expect that the intercept mode is sufficient because it provides continuous captures of all the DRAM changes made to the interested kernel space. However, in cases where one wants to support on-demand read accesses to the DRAM modules from the programmable RISC core, our solution provides a probing mode.

- In probing mode, the RISC core can issue DRAM read requests to the probe request interface (see Fig. 3.2). The requests will be converted into DRAM commands, sent to either the local DRAM modules or merged with the SB frames opportunistically. The data returned from the local DRAM modules or the NB frames received from the southerly DIMMs will be forwarded to the probe request interface and transmitted to the private DRAM using a DMA engine. Probing mode requires no SB commands from the memory controller, otherwise it will cause conflict because the host memory controller is not aware of the DRAM commands issued from the probe request interface. In probing mode, it is preferred that the memory controller does not send DRAM commands to the AMB. This can be achieved by putting the host CPU in an idle

state (discussed in Section 4.3). When SB DRAM commands are received, and there is an outstanding read request from the RISC core with already issued DRAM commands, the result is a collision. The extended AMB contains logics for detecting such collisions and responding to them when they occur.

The extended AMB does not add any extra delays to the critical paths of NB and SB frame transmission (marked as dashed lines in Fig. 3.2). Our solution only requires the split of NB and SB frames into FIFOs where they will be analyzed and preserved according to the policies set by the programmable RISC core. The split action involves only forwarding a copy of the incoming/outgoing frames thus incurs no additional latency. For each AMB, it can have upto 4GB DRAM modules connected locally. The size of the private DRAM is much smaller (e.g., 256MB). Furthermore, only the first FB-DIMM connecting to the memory controller needs the extended AMB. The rest southerly DIMMs can use the regular AMB.

The RISC core boots from a system image stored in the private flash memory attached to the AMB. The system image contains a lightweight OS together with the necessary detection software and data. The entire system is fully transparent to the host processor. To the host, the AMB appears as a regular fully buffered DIMM. The solution is completely stand-alone. No hardware or software modification is required to an existing host.

Chapter 4

Architecture Design

This chapter describes the architecture design of MGuard system. We mainly rely on extending the hardware on existing chip AMB, Adavanced Memory Buffer which forwards the traffic, to intercept the signal sent by memory controller and request the value from certain virtual address. For some crucial design decisions, brief discussions will be given. First in Section 4.1, we describe the interception behavior of the MGuard when memory controller communicate with DRAM by using packets. Then, in Section 4.2, we discuss how to get the virtual address from the signal packets by address mapping. In Section 4.3, we provide a solution which MGuard system could check the kernel value actively. Section 4.4, we discuss how the system could continuously monitoring the kernel behavior without interfering the system performance. Finally, we give a simple example of how MGuard work in Section 4.5.

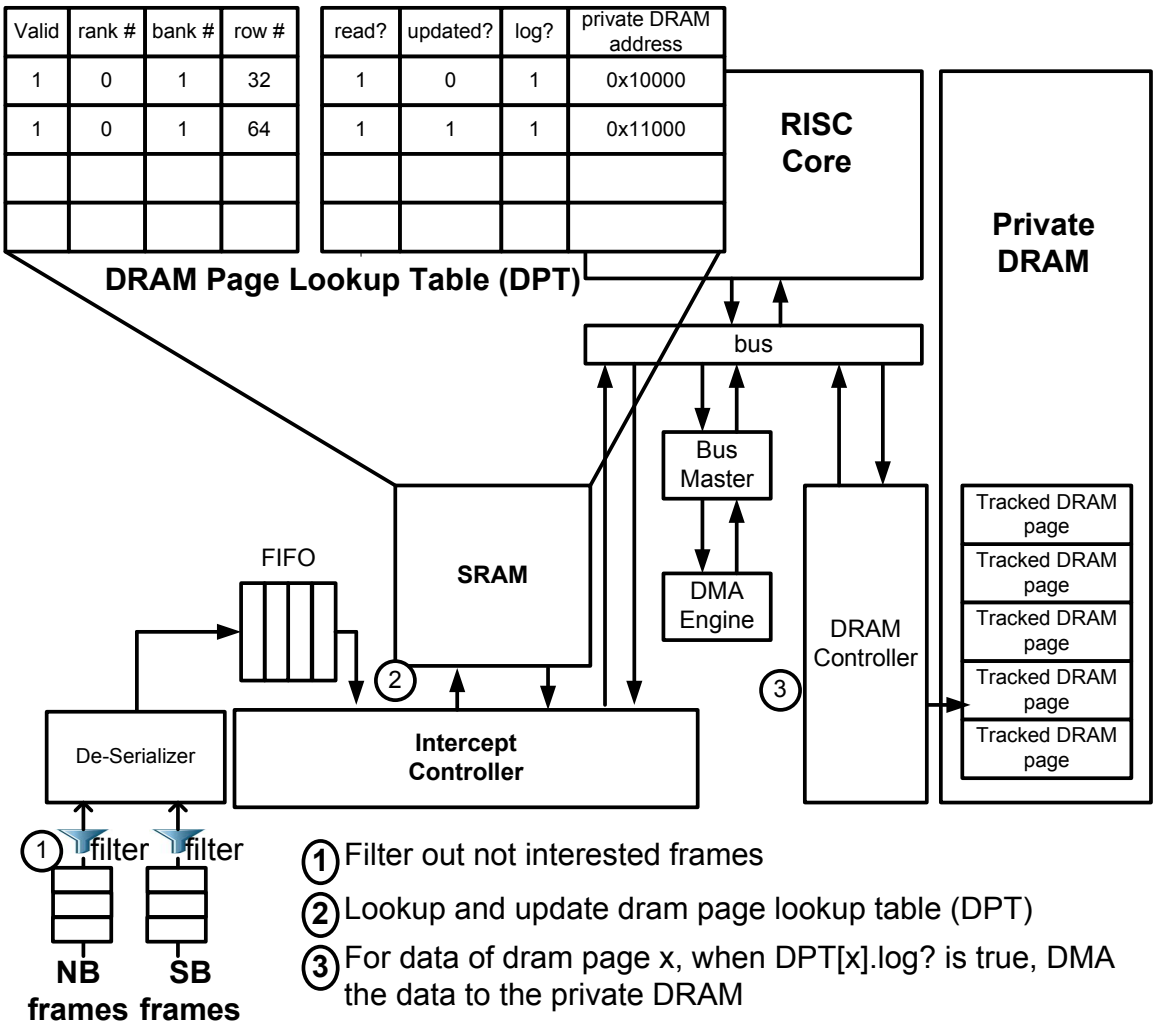

Figure 4.1: Frame Interception Datapath

4.1 Frame Interception

The detailed design of our frame interception is presented in Fig. 4.1. The FB-DIMM system uses a high speed serial packet-based protocol to communicate between the memory controller and the DIMMs. Frames may contain data and/or commands. Commands include DRAM commands such as row activate (RAS), column read

(CAS), precharge command and so on, as well as channel commands such as commands for initialization and synchronization. The commands are transmitted over southbound (SB) channel that supports a frame rate at the DRAM clock frequency (frame period equal to the DRAM clock interval). Depending on the row buffer management policy and state of the DRAM memory system, the memory controller translates each memory transaction into one or more DRAM commands such as RAS, CAS and precharge. For example, in an open page memory system, a memory transaction could be translated into: a single column access command if the row is already open; a precharge command, a row access command and a column access command if there is a bank conflict; or a row access command and a column access command if the bank is currently idle.

The SB frame contains bits for identifying the command type, rank number, and bank number. If the command is a row activation, it must include the row address; and if it is a read or a write command, it must include the column address. If the command contains write data, each frame can have 72 bits of write data. Data returned from the DIMM are sent back to the memory controller through the northbound channel with a frame rate also at the DRAM clock frequency. An NB data frame can contain two 72 bit data chunks. Other NB frames are idle, alert, or status frames.

In our extended AMB, copies of NB and SB frames are forwarded to a FIFO where they will be analyzed. Before getting admitted into the FIFO, uninterested frames such as channel frames, idle frames, and NOP frames are filtered out. This reduces the number of frames to be analyzed. A frame intercept controller reads frames

from the FIFO and matches them against a SRAM lookup table, called DRAM page lookup table. The lookup table is organized into multiple rows. Each row contains a tag field comprising a valid bit, a rank number, a bank number, a row address, and a data field (see Fig. 4.1). The lookup table rows can be programmed and configured by the RISC core. The SRAM is dual-ported. It allows access from both the RISC core (through a SoC bus) and the intercept controller.

The intercept controller also monitors the flow of the captured DRAM commands. It will check if the DRAM commands operate on data of interested DRAM pages according to the DRAM page lookup table. If a row with matching tag field is found and the valid bit is set, the intercept controller may update the row's data field and save the DRAM data to the private DRAM according to the settings in the data field. For each row, the data field contains both dynamic data modifiable by the intercept controller and static data. The dynamic data include a bit for tracking whether a DRAM page is modified (updated?), and a bit for tracking whether a DRAM page is read (read?). There is a bit (requireLog?) indicating whether the captured data of a DRAM page should be saved to the private DRAM. The static data contain a private DRAM physical address that points to the location where captured data of a DRAM page should be stored. If requireLog? is set, the intercept controller will transfer the captured data to the private DRAM using a DMA engine that also connects to the shared SoC bus. Since the private DRAM has smaller size, only DRAM pages of important kernel states should be kept in the private DRAM. However, the DRAM page lookup table can monitor states of a much larger memory space and track access history of its DRAM pages using the read? and updated?

bits. For example, the lookup table may have 16K rows that can be used for tracking states of 16,384 DRAM pages.

In our approach, frames are captured and analyzed without adding extra latency overhead to the critical paths, dashed lines in Fig. 3.2. This means that the extended AMB does not increase latency of DRAM accesses.

4.2 Address Mapping

In general, MGuard could get the virtual addresses of intercepted frames by two steps. The physical addresses of intercepted data will be first obtained based on the frame of commands which include the information of row activate(RAS), column read (CAS) and so on. At high level, the memory controller will generate the channel ID, rank ID, bank ID, row ID, and column ID from the physical address based on different mapping scheme and send them as command packets. The mapping scheme is also related to the memory row buffer management policy, which can be either implemented in open-page mode or close-page mode. In the open-page mode, by getting benefits from temporal and spatial locality of the address request stream, adjacent cacheline addresses can be mapped into the same row across different channels. Similarly, the consecutive cacheline addresses are mapped to different channels to minimize the chances of bank conflict in the latter mode. The parametric variables are defined in Table 4.1. In the baseline open-page address mapping scheme, the memory system can be denoted as r:l:b:n:k:z where the lower case letter is the binary logarithm of the upper case letter. The baseline close-page address mapping scheme is denoted as

	Description		Description
K	Number of channels in system	C	Number of columns per row
L	Number of ranks per channel	V	Number of bytes per column
B	Number of banks per rank	Z	Number of bytes per cacheline
R	Number of rows per bank	N	Number of cachelines per row

Table 4.1: Definition of Memory System Address Variables

r:n:l:b:k:z. When MGuard intercepts a southbound command frame, the information of row ID, column ID, channel ID, etc could be extracted. With such information, MGuard could calculate the physical address of the data based on certain mapping scheme.

Next, these physical addresses will be translated into virtual addresses by checking the mapping information from "system.map" file. For instance, in a 32-bit system, the Linux kernel usually locates in the top 1 GB of the 4GB virtual address space. Depending on the platform's memory map, this will be mapped to a physical address in the physical memory. To find out where these symbols are loaded in the main memory, subtract PAGE_OFFSET, 0xC00000000 in our example, from the symbol address to get the offset and add this offset to the starting physical address of the kernel in the physical memory as determined from the system memory map. Vice versa, one can always tell whether a physical address is in kernel space or not.

Each distribution of OS kernel has its own "system.map". MGuard can detect which OS is used by a host (e.g., through a memory-only-based OS fingerprinting [17]) and choose the corresponding kernel map. It can update and configure the DRAM page lookup table accordingly. MGuard stores kernel information for different OS kernels in the private flash memory. For a new OS distribution, its kernel map and associated data can be downloaded to a MGuard system through the serial

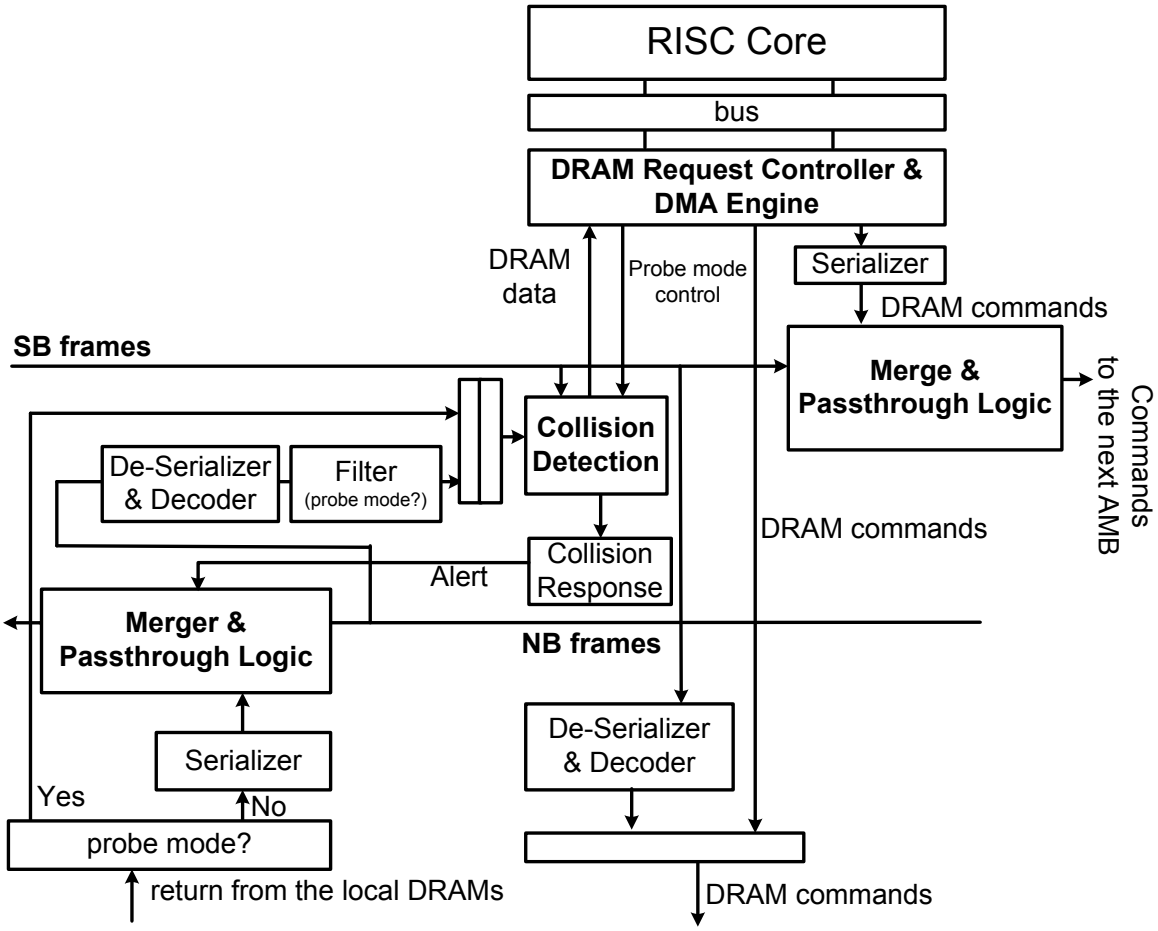

Figure 4.2: Active DRAM Probing

bus interface.

4.3 Active Probing

MGuard supports continuous capture of write data to the interested DRAM pages and reads data fetched from the interested DRAM pages. When the RISC core decides that copy of a new DRAM page should be made, it can do so by reconfiguring

the corresponding row of the DRAM page lookup table. Whenever the corresponding DRAM page is updated or read by the host CPU, a copy of the data will be stored to the private DRAM, accessible by the programmable RISC core. This is the common and recommended way for capturing the physical memory states. A

However, our solution does support a probing mode where the RISC core can send a read request to a DRAM page directly. As shown in Fig. 4.2, there is a probing request interface that can take read request from the RISC core. The probing request interface can issue DRAM commands to the local DRAM interface for reading a DRAM page or merge the DRAM commands with the SB frame traffic forwarded to the southerly DIMMs. Data returned from the local DRAM modules or from the NB frames will be forwarded to a collision detection unit. If no collision is detected, the data will be returned to the probing request interface. The probing request interface can transfer the returned data to a location at the private DRAM using a DMA engine. The private DRAM location is specified by the RISC core when it issues the read request. Probing mode requires special care because the issued DRAM commands from the probing request interface may collide with the SB commands from the memory controller. In the ideal case, when the host memory controller does not access the DRAM for enough period of time, the probing request interface can issue DRAM commands opportunistically. If the host memory controller gives sufficient slack time, everything would be fine. Otherwise, there could be collisions. The collision detection logic monitors outstanding DRAM command from the probing request interface and SB frames from the host memory controller for detecting any possible collision between them.

To ensure that there is enough time for completing DRAM commands from the probing request interface, the RISC core can temporarily put the host CPU into idle state. This can be achieved, for instance, through the system management interrupt (SMI). The system management interrupt causes the CPU to enter into the system management mode. Write-back caches will be flushed to enter the SMM. The SMI handler can be programmed to drain commands to the FB-DIMMs and hold the memory controller for sufficient amount of time before the probing request interface completes needed read accesses to the DRAM modules. SMM cannot be masked or overridden which means that an OS has no way of avoiding being interrupted by the SMI. AMB connects to one of the GPIO pins and uses it for raising a system management interrupt.

The extended AMB contains collision response logic for handling collisions whenever they are detected. For doing so, it leverages the existing fault handling mechanism of the FB-DIMM standards [23, 22]. The FB-DIMM standards support recovery from a transient failure or corrupted commands on the SB channel through alert frames sent back to the host memory controller on the NB channel. The extended AMB takes advantage of this feature and sends back alert frames when a collision is detected. To the host memory controller, the collision appears as a transient failure in the SB transmission or a corrupted SB command. In response, the host will issue a soft channel reset command to acknowledge the receipt of the alert frames and reset the command state of the AMB. The AMB receives the soft channel reset command and resets its internal command state. After the soft channel reset command, the

host may issue a sequence of commands to clear all the DRAM devices such as issuing a precharge all command to all the ranks. After that, the host memory controller can issue new SB commands. This means that a read request from the RISC core could fail if it collides with the SB DRAM commands from the host CPU and the collision causes a soft channel reset.

Furthermore, the same data path can be used for correcting kernel memory states if the RISC core is allowed to issue DRAM updates. Such operations are feasible under the current solution framework. In a read-only mode, the RISC core only sends DRAM read requests. In a second free mode, the RISC core is allowed to correct kernel data structures using the same request interface. However, great care must be taken to avoid memory state corruption. When updates are issued, the host is put into an idle state through SMM interrupt.

4.4 Kernel State Monitoring

Integrity checking consists of detecting unauthorized changes to kernel components and data structures in the volatile DRAM memory. The rationale is that kernel control data (such as system call table) tends to be static once a kernel is compiled and loaded into memory. Any dynamic modification to the kernel control data is deemed malicious. Many hypervisor or hardware-based kernel rootkit detections are based on this observation [32, 2, 39, 15].

In MGuard, the RISC core can run programs for checking integrity of the kernel space by examining the captured DRAM page states and the copied data of DRAM

pages. In addition, the extended AMB includes a serial interface that can transmit the captured data to a centralized place where the captured data can be further analyzed in detail for detecting rootkits or malware, see Fig. 3.2. The serial link also allows administrators to upload new programs or data to the RISC core so that the monitoring and checking software executed by the RISC core can be customized according to the installed system, threat contexts, and knowledge of the attacks and risks. This is one of the unique benefits of our MGuard.

The RISC core is in idle state most of the time when DRAM data are captured. It wakes up from the low power idle state in response to the external connection (via serial link) or periodically to run pre-installed software that performs routine integrity check of the captured system memory image. Since the SRAM DRAM page lookup table contains status history for the DRAM pages of interested kernel space, the RISC core can first check the tracked states of the DRAM pages storing the important kernel data structure (e.g., IDT, syscall table, interrupt handlers' code). This can be done by reading the dual-ported SRAM. If nothing is suspicious, the RISC core can go back to the low power idle state and wake up later to repeat the same routine procedure. If something is wrong, the RISC core will further examine the captured DRAM page data stored in the private DRAM. The RISC core maintains a list of hash codes (e.g., md5) for the important kernel texts and handlers (e.g., interrupt, syscall, and SMI). The sizes of the handlers are pre-determined for a distribution of operating system based on dis-assembled handler codes. For each handler, its integrity can be verified by comparing the stored hash code with the computed hash code using the captured handler data. Note that the total size of

the pre-computed hash values is very small because only kernel texts and handlers are checked. The pre-computed hash values are stored in the private DRAM and the exact size depends on the kernel version. The pre-computed hash values can be downloaded to the MGuard DIMM using serial connection. The threat model excludes insider attacks, and there is no attack surface for outsider attackers.

Different from the snapshot-based approaches, accesses to the physical DRAM is monitored continuously from the very beginning when the system boots. This allows the extended AMB and the integrated RISC core to create a baseline database of the kernel components and structures. Different from the periodic-based approach, every bit stored to the critical data components of the kernel space can be monitored and captured. The extended AMB can record the kernel data structures when they are written first time to the DRAMs. Any modifications to the kernel by the rootkits later can be detected. Furthermore, since the inspection capability is transparent to the system and integrated with the AMB that is the closest logic device connecting to the DRAM modules, it is ensured that the view of physical memory wouldn't be affected by firmware rootkits or controller rootkits. The specific space layout information for each system can be downloaded to MGuard DIMM via the serial connection. It is a significant advantage over the prior solutions.

Meanwhile, the solution can be applied for capturing SMM rootkits [11, 13], one of the most elusive types of rootkits to catch because they are stored in SMRAM (system management RAM). In SMM, SMI handlers are stored in SMRAM that is out of the reach of the host OS after they are configured. However, the SMI handlers are stored in the physical RAM, any modification to the handlers can be tracked by

the extended AMB and detected by the integrated RISC core.

Finally, thanks to the programmable feature of our FB-DIMM, MGuard also supports more complicated rootkit detection and recovery. For instance, the captured DRAM data can be also applied for cross-view based rootkit/malware detection. The idea behind the cross-view-based detection is that if a host system is infected with rootkits, it is probably hiding things and presenting a false image of the kernel data components. The view of kernel physical memory captured by the extended AMB can be compared with a common view of the system memory recorded using conventional approach. Any difference between these two views can be detected and used for revealing the rootkits hidden in the system. Our solution provides such opportunities for experimenting new rootkit/malware detection solutions. In addition, MGuard also performs certain data recovery. For instance, if a rootkit contaminates a system call table entries, we can even recover the contaminated values with the predetermined one.

4.5 An Example

To illustrate our technique, we use an example to describe how MGuard protects the integrity of the system. When we power up the system, the introspection code runs in the extended RISC core in AMB. For example, when the host CPU writes back to the DRAM, the command and data will be packed and sent by the memory controller. This packet will first go through the AMB prior to the DRAM modules. As described in Section 4.1, MGuard is able to intercept the frame whenever data

are transferred between the host CPU and the DRAM. The physical address will be extracted at this stage. As an extended hardware outside CPU, MGuard has no access to the paging files that manage the mapping between the virtual address space and physical memory address. However, the page global directory (PGD) has strong signatures [27] and we can actually search for PGDs in the physical memory as demonstrated in [17]. As such, guided by the virtual address of specified in "/boot/System.map" (for Linux kernel) file, which is a look-up table between symbol names and their addresses in memory, with the identified PGDs, MGuard can check the integrity for all the key kernel symbols such as system call table, ITD, and many other kernel function pointers as demonstrated in Chapter 5.

Chapter 5

Security Analysis

MGuard provides a holistic, off-CPU solution in detecting any rootkits outside the hardware ring. To demonstrate this feature, we developed a kernel rootkit detector, a hypervisor rootkit detector, and a SMM rootkit detector using the md5 approach mentioned in Section 6.4. In this experiment, we developed a functional simulator for MGuard that is integrated with the Bochs full-system emulator. The analysis environment supports a complete operating system with rootkit installed. In the following, we report our experiment results.

OS Kernel Rootkit Detection We took 12 source code available rootkits from *packetstormsecurity.org* and tested them with MGuard using Linux kernel 2.6.08. Not surprisingly, as presented in Table 5.1, MGuard performs incredibly well and it successfully identifies 11 rootkits that tamper such as system call table (SCT), interrupt descriptor table (IDT), and the global kernel function pointers (e.g., `tcp4_seq_show`).

It fails to identify `adore-ng-2.6` because this rootkit also modifies kernel heap object but our current detection logic does not include the heap object traversal.

When detecting a kernel rootkit intrusion, there could be multiple options for the response. The most simple one is to notify the administrator and restart the system. The more complicated one is to enable an automatic response. Thanks to the programmable feature of MGuard, we can support many extra security policies such as automatic fixing of the contaminated kernel function pointers, especially those pointers with known instruction addresses. For instance, all the SCT entries are always pointing to the system call handler code with known addresses, once a kernel is compiled. We can thus prefetch these addresses before loading an OS and repair the values if they are hijacked by kernel rootkits. In our experiment, we configured MGuard to repair the IDT entry, SCT entry, and kernel global function pointers. We succeeded in recovering the values when a rootkit attempts to overwrite these pointers.

Hypervisor Rootkit Detection A hypervisor is a trusted platform in the virtual machine environment by default. However, many attacks target hypervisor to obtain a higher level privilege of either a guest OS, or even a host OS without being detected. To demonstrate that our MGuard can detect hypervisor rootkit, we developed two in-house KVM rootkits based on kvm-kmod-3.5. In particular, one KVM rootkit hijacks `kvm_vmx_exit_handler`, and the other rootkit hijacks `kvm_x86_ops`, to introduce the illicit behavior. We also used a real-world HVM rootkit that targets `vmexit` handler [8] to test MGuard. As summarized in Table 5.1, MGuard succeeded to detect all the contaminations done by the hypervisor rootkits.

Rootkit Type	Name	Attack-Vector	Detected?
OS Kernel Rootkit	hookswrite	IDT table	✓
	int3backdoor	IDT table	✓
	kbdv3	syscall table	✓
	kbeast-v1	syscall table, tcp4_seq_show	✓
	mood-nt-2.3	syscall table	✓
	override	syscall table	✓
	phalanx-b6	syscall table, tcp4_seq_show	✓
	rkit-1.01	syscall table	✓
	rial	syscall table	✓
	suckit-2	IDT table	✓
	adore-ng-2.6	global and heap pointers	✗
	synapsys-0.4	syscall table	✓
Hypervisor Rootkit	Kvm-kmod1	`kvm_x86_ops`	✓
	Kvm-kmod2	`kvm_vmx_exit_handler`	✓
	HVM-rootkit	`vmexit_handler`	✓
SMM-Rootkit	VMBR	Redirecting IRQ	✓
	SMM-Reload	Cache Poisoning	✓
	BIOS-Rootkit	-	✓

Table 5.1: OS Kernel, Hypervisor, and SMM Rootkit Detection Using MGuard.

SMM Rootkit Detection Because SMM has its own memory space (called SMRAM) and all memory accesses to SMRAM are arbitrated through the Memory Controller Hub (MCH), it can be made invisible to code running outside of the SMM. Therefore, it is impossible to detect SMM rootkits by using devices residing off the RAM (e.g., [32]). However, we can track and detect any modification to the SMRAM (i.e., SMM rootkit) using our extended AMB and the integrated RISC core. To show that our MGuard can detect SMM rootkit, we first implemented a SMM-based keylogger and network backdoor (described in [13]) to enable keystroke logging in SMM and send the logged keystroke to another machine via UDP. The

rootkits were tested using both emulation and real hardware. Bochs supports complete emulation of SMM functionality. We also tested the rootkit by injecting it into the BIOS of ASUS P5Q based on Intel P45 hardware by using Windows kernel driver. Encouragingly, MGuard succeeded to detect all these malicious code inside SMRAM. In addition, as shown in Table 5.1, MGuard succeeded to detect other kinds of SMM rootkit such as BIOS rookit as well.

Discussion MGuard does have certain limitations. Because the program for monitoring kernel space is running on the programmable RISC core, it cannot access certain states internal to the processor. In addition, because MGuard only monitors changes to the physical memory, it is hard to detect attacks that leave no trace in the physical memory. However, this can be addressed by flushing the on-chip caches periodically through SMI interrupts sent from MGuard. In other words, what the program can observe (e.g., memory pages) may not be what we want to observe (e.g., processes and context switch). However, because all data required to be processed by CPU should be eventually located in the main memory if caches flush frequently, the attacks will be detected within short time. Also, in our current prototype, we only demonstrate that we can identify the rootkits that tamper with kernel global function pointers. While our experiment shows we failed to identify `adnore-ng-2.6` rootkit, MGuard can actually identify the pointers in kernel heap as well. For instance, we can integrate other techniques such as KOP [9], SigGraph [27], or OSck [21] to traverse the kernel heap and detect the rootkit. We leave this as one of the future efforts.

Chapter 6

Performance Evaluation

The memory integrity checking is off CPU critical path. We evaluate the performance, energy, and area overhead of our MGuard based on the extended architectural and DRAM simulators. In particular, we extended three simulators (i.e., GEM5 [5], DRAMSim2 [34], and OR1ksim [1]). GEM5 is a system simulator built from a combination of M5 [6] and GEMS [28] simulators. GEM5 supports most commercial ISAs such as x86, ARM, and MIPS. It can run a full system simulation and provide a cycle based model for out-of-order processors. DRAMSim2 [34] is a cycle accurate open source JEDEC DDRx memory system simulator. It provides a DDR2/3 memory system model that can be used with many architectural simulators including GEM5. DRAMSim2 can model power, latency, and bandwidth of DDR2 and DDR3. The programmable RISC core is modeled based on OR1ksim, a generic OpenRISC architecture simulator capable of emulating the OpenRISC-based computer systems. To analyze DRAM interaction while executing benchmarks, we integrated FB-DIMM

and MGuard simulation capability with GEM5. The developed FB-DIMM simulation integrates various proposed components with parameters derived from reference RTL implementations. The simulator integrates OR1ksim with DRAMSim2 by using the DRAMSim2 library mode. FB-DIMM support is modeled according to [29] and [23, 22]. The default AMB clock frequency is set at 400MHz.

6.1 Machine Parameters

We hook up the GEM5 simulator with MGuard simulator to model a quad-core system. In particular, the multicore CPU is integrated with 2MB L2 cache and an on-chip memory controller. The DRAM is modeled based on the micron FB-DIMM specification. In the GEM5 side, the simulation is performed with an out-of-order CPU model running at 2GHz and x86 ISA. The CPU model has seven pipeline stages: fetch, decode, rename, issue, execute, writeback, and commit. Each processor core has pipeline resources: branch predictor, reorder buffer, instruction queue, load-store queue, and functional units. The I-TLB and D-TLB have 64 fully associative entries. The L1-instruction and L1-data caches are 64KB write-back caches with 64-byte block size, and an access latency of 2 cycles. The L2 cache is unified, non-blocking, 2MB size, 16-way associativity, 128-byte block size, and has an 10-cycle access latency.

In the FB-DIMM side, we configure a 2GB FB-DIMM associate with the GEM5. The parameters is shown in Table 6.1. The extended AMB is modeled with the parameters shown in Table 6.1. By default it uses a fixed latency mode, which

DRAM module	Value	DRAM timing	Value
Number of channels	1	t_{CK}	2.5ns
Number of banks	8	t_{RAS}	18
Number of rows	16384	t_{CAS}	5
Number of columns	2048	t_{RCD}	5
Device width	4	t_{RC}	23
Refresh Period	7800	t_{RP}	5
AMB	**Value**	**Memory Controller**	**Value**
Passthrough lat	2.2ns	Policy	open page
Deserialization lat	8.0ns	Scheduler	FR_FCFS
Serialization lat	5.0ns	Read/write queue size	32
Channel mode	fixed lat mode	Write mode enter/exit threshold	28/6
M-Guard	**Value**	**M-Guard**	**Value**
Lookup table lat	1.9ns	FIFO size	48 entries

Table 6.1: DRAM, Extended AMB, and FB-DIMM Parameters

means, to the host memory controller, all DIMMs on the channel will appear to have a fixed latency. The DRAM is managed using an open-page policy. The FB-DIMM latency is derived from [29, 22]. The signal path from the memory controller to the first FB-DIMM takes 0.6 ns delay. The signal path from one FB-DIMM to another takes 0.2 ns (assuming 4cm distance). The round trip latency can be modeled by the simulator based on the parameters given in Table 6.1 under the fixed latency mode and open-page policy. The AMB parameters are based on the AMB standard, published numbers of commercially available AMB chips from vendors (e.g., [12]), and [29].

6.2 Benchmarks

For performance evaluation, we used the SPEC CPU2006 benchmark suite. CPU2006 is SPEC's next-generation, industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem, and compiler. SPEC designed CPU2006 to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications. These benchmarks are provided as source code and require the user to be comfortable using compiler commands as well as other commands via a command interpreter using a console or command prompt window in order to generate executable binaries. We tested ten memory intensive benchmarks of the SPEC CPU2006. These include, bzip2, gcc, gobmk, hmmer, sjeng, libquantum, h264ref, omnetpp, namd, and lbm. The detailed descriptions of the benchmarks can be found in [19]. We also choose 5 popular server benchmarks applications with real-world input dataset. The simulation started when the application passed the initialization stage. The cycle based simulation executed each benchmark application for one billion instructions or until it finished depending on which one was longer.

6.3 Synthesis

The major components of our extended AMB design include: a frame intercept controller, an integrated programmable RISC core, a DRAM controller for accessing private DDR2 memory, a shared SoC bus and a DMA engine, a SRAM lookup table,

Table 6.2: Twenty Benchmarks for Performance Evaluation

Benchmark	Description
bzip2	A popular compression utility based on Burrows-Wheeler algorithm
gcc	GNU C compiler
gobmk	Artificial Intelligence: go
hmmer	Search Gene Sequence
sjeng	Artificial Intelligence: chess.
libquantum	Physics: Quantum Computing
h264ref	Video Compression
omnetpp	Discrete Event Simulation
namd	Biology / Molecular Dynamics
lbm	Fluid Dynamics

and various FIFOs and filters. For rapid prototyping, we leveraged open source IP blocks whenever it is possible. We evaluate the power and area performance of our MGuard by integrating these components and synthesizing the design using Synopsys tools with FreePDK at 45nm. The dual-ported SRAM is evaluated using CACTI 6.0. The private memory is 256MB DDR2 and simulated using DRAMSim2.

The shared system bus is based on the open-source Wishbone bus [20]. Wishbone is defined as an on-chip internal bus for the System-on-Chip (SoC) architecture, which is a portable interface for use with semiconductor IP cores. The DDR2 memory controller for the private DDR2 is based on [4], an open-source implementation of DDR2 SDRAM controller. An open-source DMA engine compatible with the Wishbone SoC bus is used as the DMA engine for the private DRAM [38]. The Wishbone bus provides a common bus between these IP cores. The FIFOs are adapted from Verilog implementation of generic FIFOs. Verilog implementation of our design is

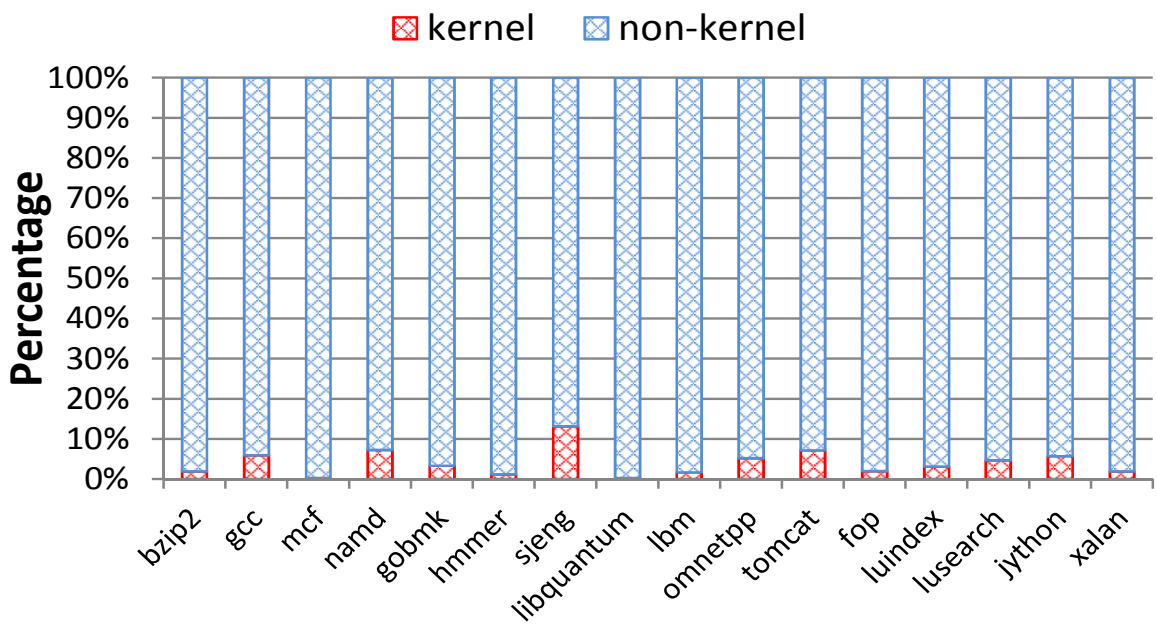

Figure 6.1: Percentage of Kernel Accesses

synthesized using the Design Compiler of Synopsys. It provides parameters for estimating overhead and tuning the cycle based simulation models.

6.4 Performance Analysis

Fig. 6.1 shows the percentage of DRAM accesses to the kernel during execution of the benchmark applications. In most cases, only about 4% of the accesses are in the kernel. Sjeng has the largest percentage of kernel accesses (i.e. 11%). The average is 4.3%. The types of kernel accesses are illustrated in Fig. 6.2. Among all the kernel DRAM accesses, only less than 15% are write accesses. All the others are read accesses. The only exception is lbm application. Almost half of the DRAM accesses are writes. The average of all the benchmark applications is 13%. In terms

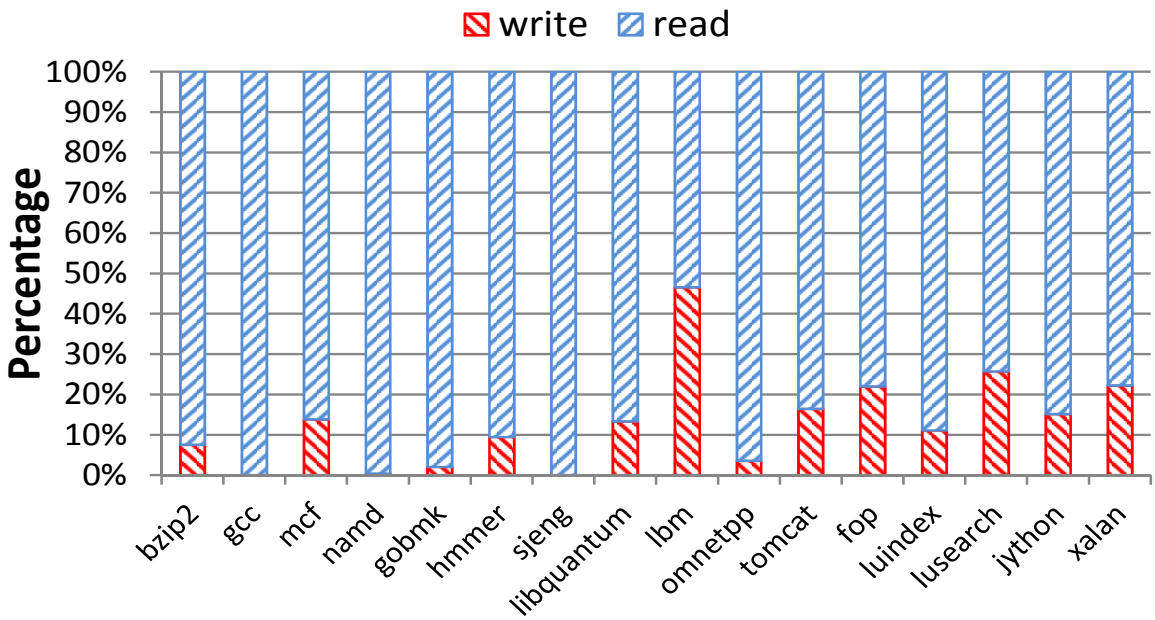

Figure 6.2: Types of Kernel Accesses

of kernel DRAM pages accessed during execution of the benchmark applications, the results are displayed in Fig. 6.3. It shows how much kernel space is touched. In most applications except lbm and omnetpp, only 2% of the kernel DRAM pages are touched. lbm and omnetpp have about 11% and 20% touched kernel DRAM pages. On average, 93.7% kernel DRAM pages are not accessed. There are far fewer updates to the kernel DRAM pages than reads. The results suggest that during execution of benchmark applications, only a small percentage of DRAM accesses are in the kernel. Meanwhile, the majority of the kernel DRAM pages are not accessed. Of these kernel DRAM pages touched, there are more read accesses than write accesses. This means that under normal application execution environment, the workload of frame interception by the extended AMB is very light.

A simulator integrated with the DRAMSim2 library is used to study how busy the

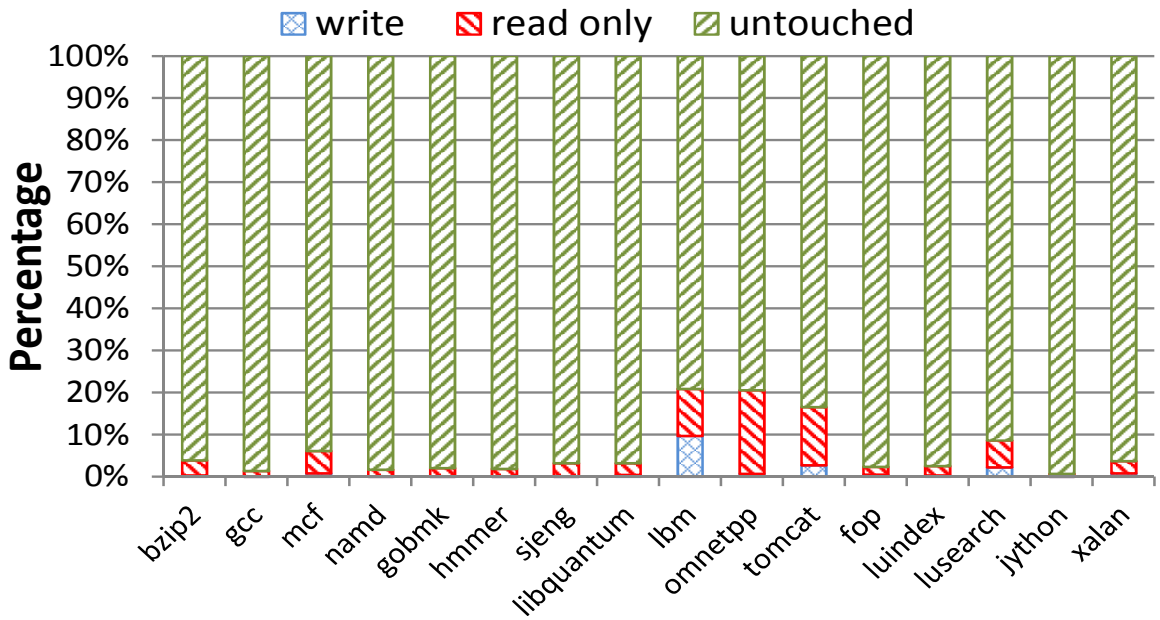

Figure 6.3: Accesses to Kernel DRAM Pages

FB-DIMM is during execution of the benchmarks. Fig. 6.4 shows the percentage of DRAM usage. According to the statistics, one can see that for certain applications, the DRAM is idle most of the time during benchmark execution. For six out of the ten benchmarks, the DRAM idle time is about 90%. Benchmark gobmk, sjeng, and lbm have more DRAM busy time. One can further break down the DRAM idle time in terms of how long the idle duration lasts. The results are shown in Fig. 6.5. According to Fig. 6.5, when DRAM is not busy, most of the idle time duration is longer than 500ns.

We developed a set of programs (written in C and compiled using OpenRISC toolchain) for detecting modifications to the important kernel components and data structures. These include hash-based checking (md5) of system call table (SCT), its handlers, interrupt descriptor table (IDT), interrupt handlers including SMM

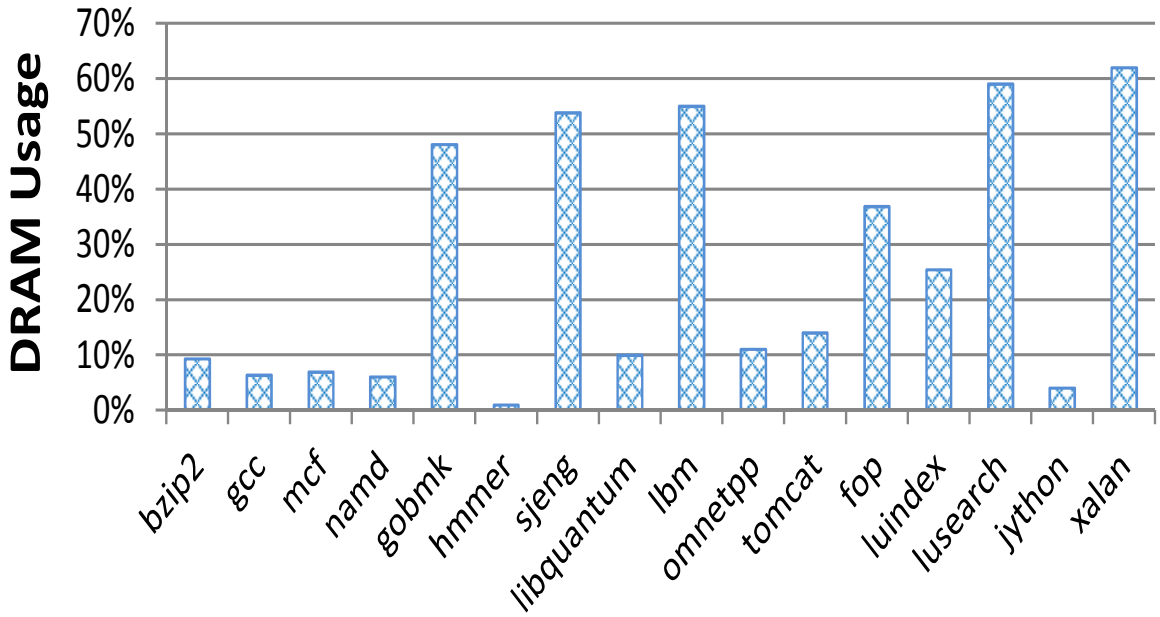

Figure 6.4: DRAM Usage for Different Benchmarks

handlers, etc. When kernel texts and handlers are changed, the modifications can be captured by the AMB on-chip frame intercept mechanism. The RISC core can compute the md5 hash code of a tracked kernel space, and compare the hash result with a known hash value pre-computed. We collected performance data on these programs using the cycle-based OpenRISC simulator integrated with the simulation environment and DRAMSim2. The results indicate that it takes about 1.3ms to check the integrity of the syscall table using md5. IDT can be checked in 0.02ms because it is much smaller than the syscall table. For checking integrity of syscall/interrupt handler, it takes on average 4.34ms using md5 hash.

There is an interesting question on whether the system can keep up with the memory throughput. In our paper, the AMB is running at 400MHz. The southbound frame interval is 2.5ns at peak. The build-in filter will remove NOP frames, idle

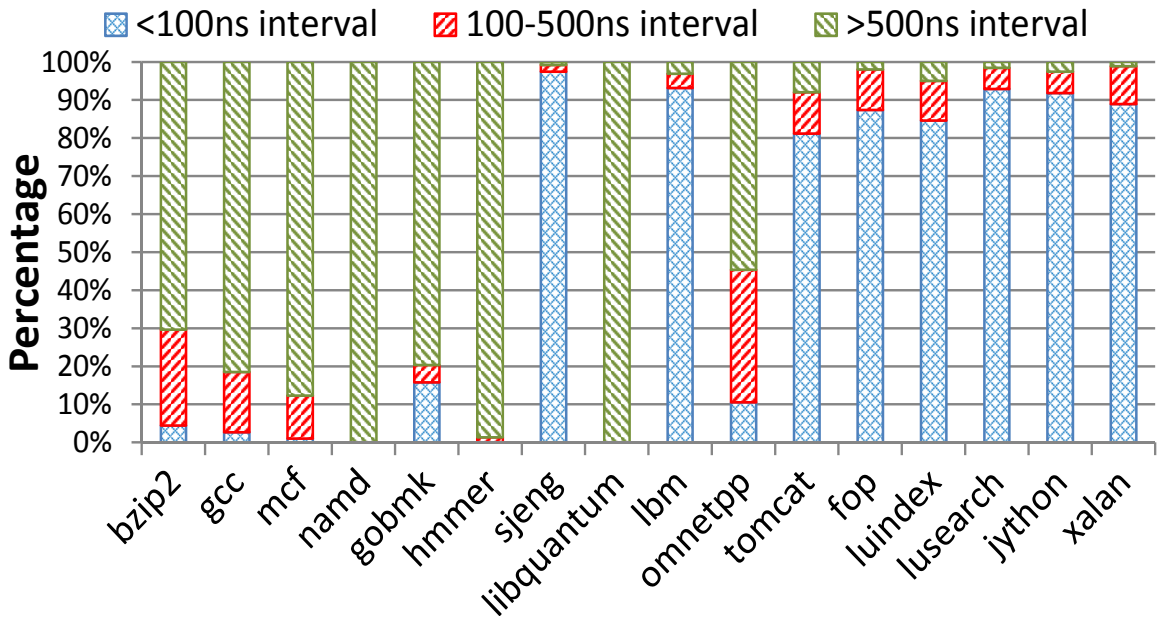

Figure 6.5: Distribution of DRAM Command Interval

frames, channel frames before they are handled by the intercept controller. The intercept controller only processes the frames accessing the kernel space. It will compare the DRAM address with data stored in a lookup table. The lookup table has access latency of 1.9ns based on Verilog implementation and synthesis results (listed in Table 6.1). This means that even in the very unlikely worst case scenario, all the frames are read/write frames at peak rate, MGuard would be able to handle the frame traffic.

6.4.1 Hardware Overhead

The area and peak power consumption of different hardware components are shown in Fig. 6.6. A fully synthesizable implementation of MGuard AMB extension at

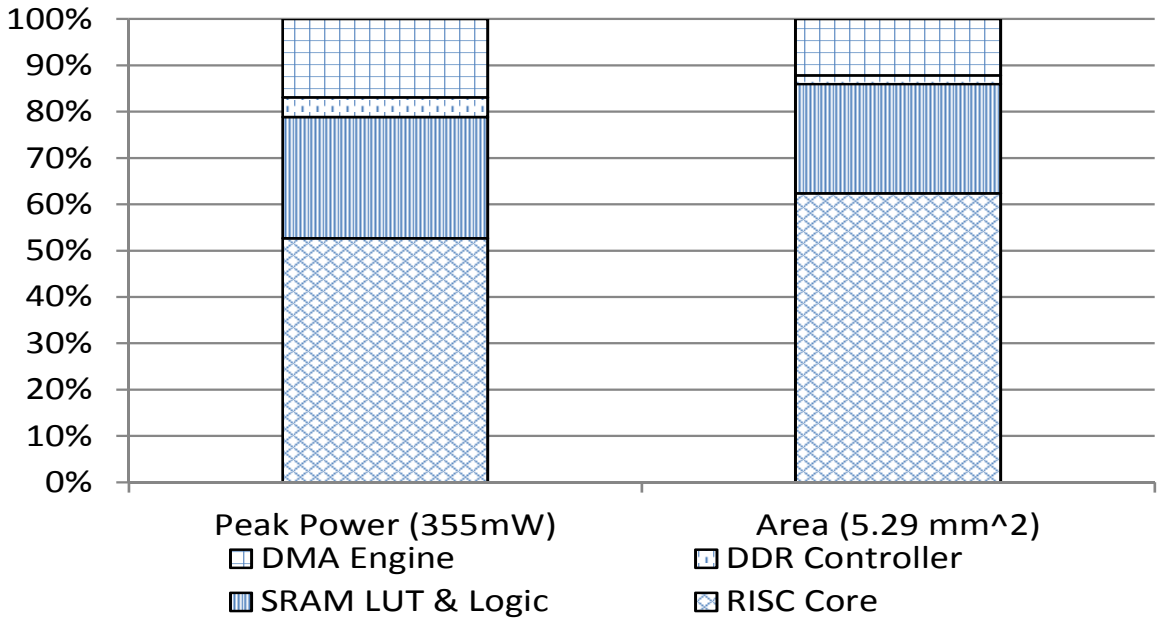

Figure 6.6: Area and Peak Power Characteristics of MGuard Extension

45nm operates at 400MHz, occupies $5.3mm^2$, and dissipates 355mW of peak power. Most of the area and power are consumed by the RISC core and SRAM lookup table. For reducing die size overhead, we optimized the RISC core by removing unnecessary components and unused facilities. The result total die size is 70% smaller than using the default OpenRISC design. Compared with the $58mm^2$ die area of Intel 6400/6402 advanced memory buffer, the total overhead of die size is only about 9.1%.

In FB-DIMM architecture, the AMB can consume around 6W of power based on vendor reported measurements and FB-DIMM standards. A large part of the power consumption can be attributed to the high speed serial links. In our extended AMB, it contains several on-chip components that may increase the overall AMB power consumption. Power models of these components are created according to the descriptions in Section 6.3. A detailed transaction-based power model for AMB is

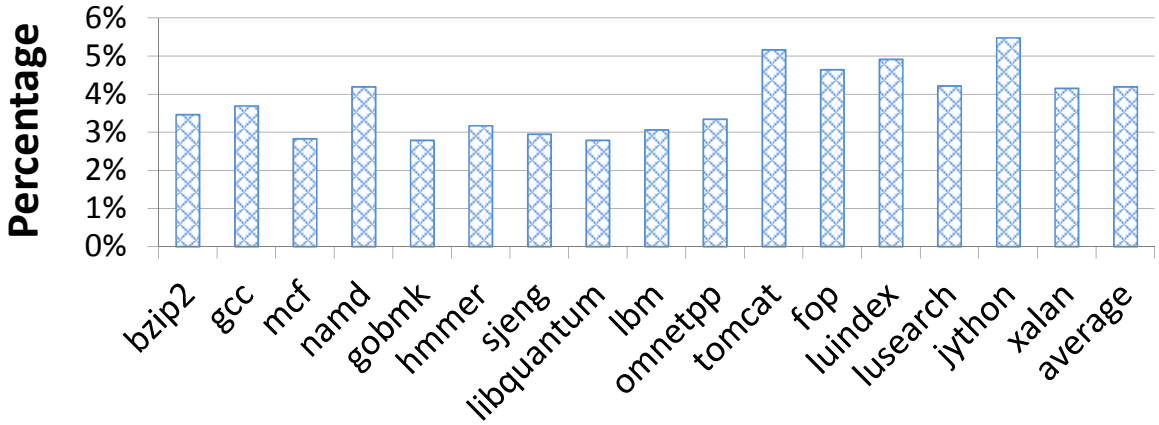

Figure 6.7: AMB Relative Power Overhead

constructed according to the AMB standards, vendor datasheets, and previous published results [45]. Power consumption of the DRAM page lookup table is modeled using Cacti 6.5. For each benchmark, the DRAM page lookup table power consumption is simulated using DRAM commands captured by DRAMSim2. Only for tracked kernel DRAM pages, the SRAM lookup table is updated. The RISC core wakes up periodically. In rest of the time, it stays in low-power dozy mode. For the benchmarks, the introduced AMB power overheads are shown in Fig. 6.7. For all the applications, the AMB on-chip power overhead is rather small. The maximum is less than 5.5% and the average is about 4.2%. In addition to the AMB power overhead, we also evaluated the overall power overhead considering the FB-DIMM as a whole (counting all the DRAM devices accessible by the host memory controller and the private DDR2 DRAM devices). Power modeling of the DRAM devices is given by DRAMSim2. The private DRAM is accessed by the frame intercept controller and the RISC core. The overall FB-DIMM power overhead is shown in Fig. 6.8. On average, the power overhead when considering the FB-DIMM as a whole is only 4.4%.

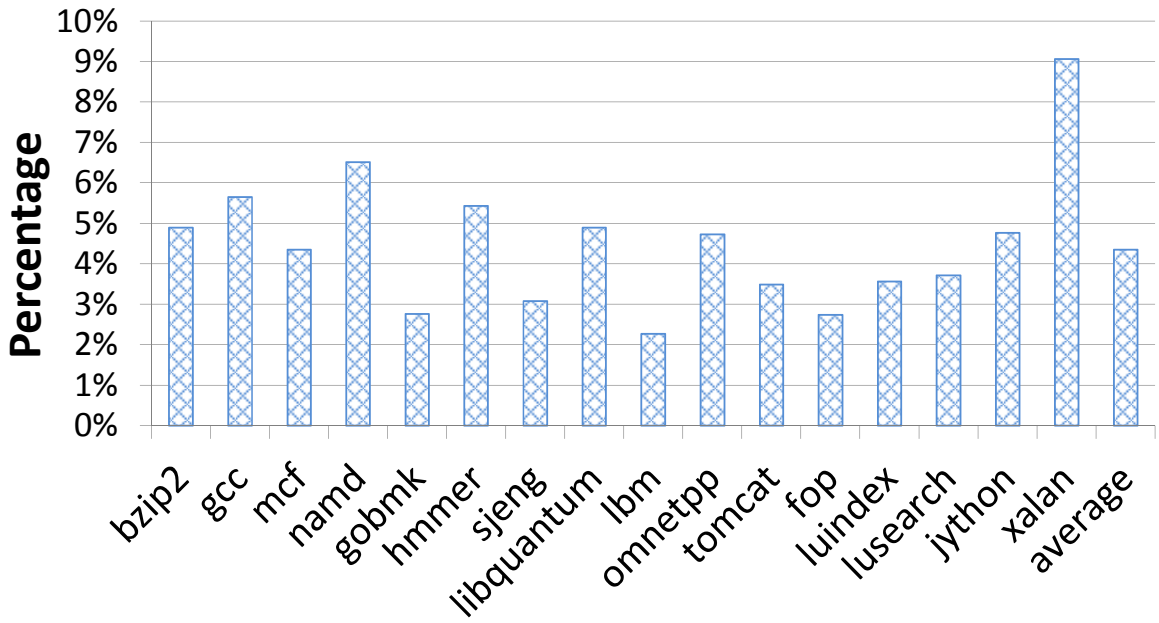

Figure 6.8: Total FB-DIMM Relative Power Overhead

For named, hmmer, and xalan, the overhead is over 5%.

Chapter 7

Conclusion and Future Work

We have presented MGuard, a new hardware-assisted, most inner ring system integrity monitor integrated with AMB DRAM. The distinctive feature of MGuard is that it continuously checks the integrity of all the outer ring memory access including OS kernel and hypervisor of interest, off the DRAM critical path. It has no performance overhead and consumes on average 3.5% more power according to our simulated experimental results. We have proven with real-world rootkits that MGuard can effectively detect 11 OS kernel rootkits, 3 hypervisor rootkits, and 3 SMM rootkits with our kernel rootkit, hypervisor rootkit, and SMM rootkit detectors without any false positive or false negative. Our MGuard is entirely transparent to all the outer ring software and hardware, and can therefore be easily applied to commodity systems.

As discussed in Chapter 3, the protection of MGuard is based on the static kernel addresses. In order to add/remove the kernel addresses which need to be protected,

MGuard can update the protection list via the serial connection. This scheme requires our MGuard connected to a clean host which will not tamper MGuard. The address mapping scheme is not accurate under multiple channels situation. Thus, we need further discussion of specific synchronization method between different memory channels. To make the MGuard works as we expected in terms of timing, the next step would be measure the peak throughput of OpenRISC core. Last but not the least, we notice that FB-DIMM is not a popular DRAM in the market. How to apply this technique to the latest DRAM technologies such as DDR4 would be an interesting question. In the near future, we will think about how to adopt MGuard to DDR4.

Bibliography

[1] OpenRISC 1000: Architectural simulator. `http://www.opencores.org/openrisc,or1ksim`.

[2] 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 *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[3] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*, pages 77–86, Anaheim, California, December 2008.

[4] U. Becker. Ddr2-sdram controller.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39:1–7, Aug. 2011.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July 2006.

[7] Buffer. Hijacking linux page fault handler. *Phrack Magazine*, 0x0B, 0x3D, Phile #0x07 of 0x0f, 2003.

[8] Y. Bulygin and D. Samyde. Chipset based appraoch to detect virtualization malware aka deepwatch. *Black Hat Security Conference*, 2008.

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference*

on Computer and Communications Security (CCS'09), pages 555–565, Chicago, IL, USA, 2009.

[10] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 566–577, Chicago, Illinois, USA, 2009. ACM.

[11] L. Duflot, D. Etiemble, and O. Grumelard. Using cpu system management mode to circumvent operating system security functions. *DCSSI 51 bd. De la Tour Maubourg 75700 Paris Cedex*, 2007.

[12] Elpida. Fully buffered dimm - main memory for advanced performance. *www.elpida.com*, 2004.

[13] S. Embleton, S. Sparks, and C. Zou. Smm rootkits: a new breed of os independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks*, SecureComm '08, pages 11:1–11:12, 2008.

[14] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 586–600, Washington, DC, USA, 2012. IEEE Computer Society.

[15] Y. Fu and Z. Lin. Exterior: using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '13, pages 97–110, New York, NY, USA, 2013. ACM.

[16] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium*, 2003.

[17] Y. Gu, Y. Fu, A. Prakash, Z. Lin, and H. Yin. Os-sommelier: memory-only operating system fingerprinting in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 5:1–5:13, New York, NY, USA, 2012. ACM.

[18] J. Heasman. Implementing and detecting a pci rootkit. *White Paper of Next Generation Security Software Ltd.*, 2007.

[19] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[20] R. Herveille. Wishbone system-on-chip (soc) interconnection architecture for portable ip cores, rev. version: B4. *Open Cores Organization*, 2010.

[21] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 279–290, Newport Beach, California, USA, 2011.

[22] JEDEC Standard. Fbdimm advanced memory buffer (amb). 2007.

[23] JEDEC Standard. Fbdimm: Architecture and protocol. 2007.

[24] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 128–138. ACM, 2007.

[25] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 91–100, Seattle, WA, USA, 2008. ACM.

[26] S. T. King, P. M. Chen, Y. min Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, pages 314–327, 2006.

[27] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, pages 41 – 46, 2011.

[28] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, Nov. 2005.

[29] R. Marwan. Fbsim and the fully buffered dimm memory system architecture. *Master of Science Thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park*, 2005.

[30] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, Glasgow, Scotland UK, 2008.

[31] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 28–37, New York, NY, USA, 2012. ACM.

[32] N. L. Petroni, J. Timothy, F. Jesus, M. William, and A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *In Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[33] 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 *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[34] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10(1):16–19, Jan. 2011.

[35] J. Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition tools. In *Black Hat Security Conference, USA*, 2007.

[36] J. Rutkowska. New blue pill. *http://www.bluepillproject.org/stuff/nbp-0.11.zip*, Aug 2007.

[37] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 437–450, 2012.

[38] R. Usselmann. Wishbone dma/bridge ip core.

[39] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: a hardware-assisted integrity monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, RAID'10, pages 158–177, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[41] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and Communications Security*, CCS '09, pages 545–554, Chicago, Illinois, USA, 2009.

[42] F. Wecherowski. A real smm rootkit: Reversing and hooking bios smi handlers. *Phrack Magazine*, 0x0d, 0x42, Phile # 0x0B of 0x11.

[43] J. Wei, B. D. Payne, J. Giffin, and C. Pu. Soft-timer driven transient kernel control flow attacks and defense. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 97–107, 2008.

[44] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat Security Conference, USA*, 2008.

[45] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. Decoupled dimm: building high-bandwidth memory system using low-speed dram devices. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 255–266, New York, NY, USA, 2009. ACM.