# CONTINUOUS MONITORING METHODS TO ACHIEVE RESILIENCY FOR VIRTUAL MACHINES

BY

CUONG MANH PHAM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

# ABSTRACT

This dissertation describes monitoring methods to achieve both security and reliability in virtualized computer systems. Our key contribution is showing how we can perform *continuous monitoring* and leverage information across different layers of a virtualized computer system to detect malicious attacks and accidental failures. For monitoring software running inside a virtual machine, we introduce HyperTap and Hprobes, which are out-of-VM monitoring frameworks that facilitate detection of security and reliability incidents occurring inside a VM. For monitoring the hypervisor, we introduce hShield, a Control-Flow Integrity (CFI) enforcement method to detect VM-escape attacks. HyperTap, Hprobes, and hShield create a complete chain-of-trust for the entire virtualization software stack.

# ACKNOWLEDGMENTS

First and foremost, I'd like to thank my advisers, Prof. Ravishankar K. Iyer and Prof. Zbigniew Kalbarczyk. Prof. Iyer gave me the direction to pursuit the research presented in this dissertation. Over the years, he also taught me how to present my ideas and my work in a convincing way. This is one of the most valuable skills that I have learned during the course of this PhD training. Prof. Kalbarczyk walked me through technical challenges to complete the research. He is always willing to listen to my ideas. Our in-depth discussions were essential to help me navigate through all the details and complexity of the research. I appreciate that despite many hurdles during the years working together, neither of them ever lost patience with me.

I wish to thank every member of the DEPEND group. Thanks, Daniel Chen, for being my first and my best American friend. I always think that you are the gift your God sent me to help me while I am in the US. You were always there to help me with technical work, and to pray before every important event in my life. Thanks, Keun Soo Yim, for teaching me how to do experimental research. Thanks, Phuong Cao and Zachary Estrada, for being my co-authors. You guys are the best collaborators that I have ever worked with. Thanks, Heidi Leerkamp, for all your support and coordination. Thanks Hui, Arjun, Subho, Homa, Skylar, and Antonio for all the discussions; I miss our group meetings.

Finally, my immense gratitude goes to my family. My parents are always my biggest supporters, they help me on every step of my life. I am lucky to walk together with my wife on our PhD journey. I treasure every moment that we share, from the coldest days of Illinois winter, to the most beautiful sunset in Maine. Her love, her support, and her meals fuel me every day. I am truly grateful to have my daughter, Joyce, come to our lives. She makes us smile more, love each other more, and work harder.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

*This chapter introduces the topic of this thesis: continuous monitoring in virtualized computer systems. We begin by briefly explaining the significance of this research, and then discuss the limitations of the state-of-the-art virtual machine monitoring techniques. We conclude the introduction by summarizing the our research objectives and contributions.*

This dissertation presents three new methods to provide low-cost and effective continuous monitoring across all software layers of a virtualized computer system. Our monitoring methods create a complete chain-of-trust for the entire virtualization software stack: from user applications and operating system running inside a virtual machine (VM) to the hypervisor running at the bottom of the stack (see Figure 1.1). Our goal is that our monitoring methods be:

- *Low cost*: incur small runtime performance overhead to the target system;
- *Effective*: support a wide range of security and reliability enforcement policies; and
- *Continuous*: expose no spatial or temporal gap for failures and attacks to escape.

The next section explains the importance of this research.

## 1.1 Motivation

Monitoring is a vital part of operating a computer system. Particularly when a system is deployed at scale, the efficient automation of monitoring is key to achieving resiliency against failures and attacks. We specifically focus on the monitoring of virtualized computer systems because virtualization is a critical technology enabling the design of modern data centers.

***Why monitoring?*** Building a computer system is difficult, but reliably and securely operating one is even harder. Undeniably, computer systems fail (accidentally or maliciously) regardless of how carefully they are constructed. In this terminology, failure is either a reliability incident or a security incident. While reliability incidents are primarily caused by the increasing the complexity of computer systems, security threats increase as data stored and processed by computers carry greater and greater value.

It is a well-established design principle to treat reliability and security incidents as the norm, rather than the exception [1]. A system operates under the assumption that it can accidentally fail or be attacked at any point in time. Therefore, to produce steady and useful progress, the system needs to be monitored so that adverse incidents are detected and mitigated as quickly as possible. This is the principle that embraces high-fidelity monitoring as essential to achieve resiliency in computer systems.

Our research shares the same core proposition with this design principle: using monitoring as the main vehicle to cope with attacks and failures. We focus on the design and construction of efficient monitoring methods that can capture high-fidelity views of target systems.

***Why virtualized computer systems?*** Virtualization is the means to enable sharing and to achieve high utilization in modern data centers. In 2012, 51 percent of x86 servers were virtualized, a 13 percent increase from 2011 [2]. In addition to virtualized servers being more prevalent than non-virtualized ones, the density of VMs on each server is also increasing [3].

The primary driving force of this trend is cloud computing, which leverages virtualization on commodity hardware as the core technology to facilitate sharing. Not unlike other types of utilities, cloud computing benefits from the economies of sharing and scaling. This is because sharing greatly decreases the cost of computing resources, which in turn attracts more users and providers to join the flow.

Given the abundance of VMs, an improvement in the security and reliability of this technology will have a large impact.

## 1.2 Target System Model

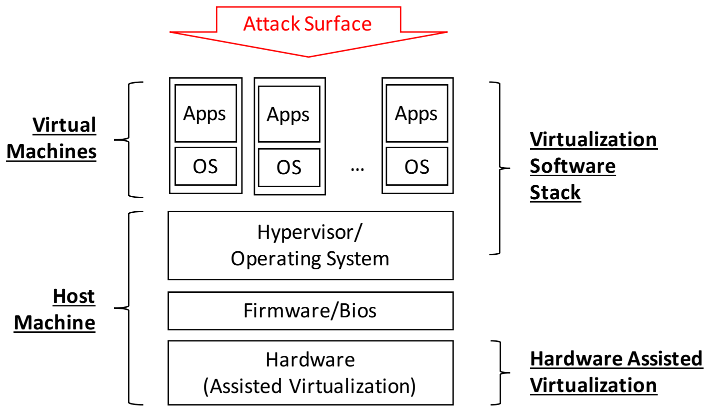

Attack Surface
Virtual Machines
Apps
OS
Apps
OS
...
Apps
OS
Virtualization Software Stack
Hypervisor/ Operating System
Host Machine
Firmware/Bios
Hardware (Assisted Virtualization)
Hardware Assisted Virtualization

Figure 1.1: A typical virtualized computer system. The virtualization software stack is the target of our monitoring.

Throughout this thesis, the target of monitoring is a virtualized system as depicted in Figure 1.1. The bottom layers, including Hardware, Firmware/Bios, and Hypervisor/OS, constitute the *host machine*. The layers on the top, including Application and OS, constitute the *virtual machines*. The host machine can accommodate multiple VMs running at the same time. From the perspective of users, VMs operate independently of each other.

Here we use the term *VM monitoring* to indicate any monitoring method that has the *protection target* (or *target* for short) in a layer of the virtualization software stack, including software running on a VM and the hypervisor. When the context is unclear, we use a more descriptive term to indicate the target of monitoring. Specifically, we use *guest operating system monitoring*, or *guest application monitoring* to indicate that monitoring targets are an operating system (OS) and applications running inside VMs, respectively. Similarly, we use *hypervisor monitoring* to indicate that the hypervisor is the target of monitoring. In addition, we use *out-of-VM monitoring* to indicate monitoring techniques deployed outside of target VMs to monitor software running inside VMs (e.g., monitoring is done from the hypervisor or from other VMs).

In the designs of our monitoring, we assume that hardware assisted virtualization (HAV), such as Intel VT-x [4] or AMD-V [5]), is an integral component of the system, and is utilized by hypervisors to implement virtualization. At the moment, all server-grade x86 processors on the market support HAV. Furthermore, all popular hypervisor implementations, such as the VMWare hypervisor family, KVM [6], XEN [7], and Virtual Box, can utilize HAV to execute VMs.

With regard to security monitoring, our threat model assumes that VMs are the *attack surface* of the target virtualized system. This assumption is derived from the model of data centers that rely on virtualization to serve users and process workloads. *Infrastructure as a Service* (IaaS) in cloud computing is a typical example of this model. In such a system, a *user* can execute arbitrary software, from user applications to their own OSs, inside VMs. Meanwhile, they do not have direct access to the host machine, except via the VM-hypervisor interface provided by HAV (details are provided in Chapter 2, which reviews virtualization and HAV background). Furthermore, we explicitly trust the underlining hardware. We also do not consider physical tampering and inside attackers (e.g., malicious administrators who already have remote access to the host machine).

In this threat model, we consider two broad scenarios: attacking a VM and attacking a hypervisor. The first scenario refers to attacks that aim at compromising software running inside a VM. Since in a typical data center setup most VMs must expose some remote access via the Internet to be used, they are constantly at risk of being targeted by attackers. The second scenario assumes the attacker has full access to a VM and exploits the VM-hypervisor interface to launch attacks against the underlining hypervisor (and other co-located VMs). For example, a public IaaS cloud allows any user to launch their own VMs at a very small cost. Those VMs can be used as an attack entry point to the hypervisor. Or a successful attack described in the previous scenario may grant the attacker administrative access to the exploited VM, which in turn can lead to an attack against the hypervisor.

## 1.3 Limitations of State-of-the-Art VM Monitoring

Despite the significant research effort that has been invested, state-of-the-art VM monitoring techniques still experience some fundamental limitations that dwarf their practicality. Those are limitations that leave critical gaps for failures and attacks to escape detection. Here we present limitations in regard to security and reliability monitoring.

### 1.3.1 Polling-and-Scanning Monitoring Paradigm

Most VM monitoring techniques, e.g., [8, 9, 10, 11, 12], follow the polling-and-scanning paradigm. In this paradigm, monitoring is done by *scanning* the target system at a specific *polling* interval. This paradigm is also known as *passive monitoring* [13].

There are two major limitations of the polling-and-scanning method. First, it leaves vulnerable time gaps between consecutive polling intervals. During those temporal gaps, a transient attack, which completely removes its footprint after completing, cannot be detected. We have demonstrated in [14, 15] that transient attacks can be crafted to evade VM monitors with a high chance of success. Next, this monitoring method can only scan the static state of target system, e.g., the state that is stored in RAM or persistent storage. What it misses is operational data about the activities of the target system, which is necessary to enforce many security and reliability monitoring policies.

### 1.3.2 Untrustworthy Input

The goal of monitoring is to capture and present a trusted view of target systems. This view is used at a later phase in a system's operational pipeline, e.g., enforcing a security or reliability policy. Thus, the input of monitoring must be carefully selected to faithfully represent the target system. This requirement is particularly imperative in the context of security monitoring, because attackers always proactively seek opportunities like this, which let them manipulate input to falsify monitoring views.

However, many out-of-VM monitoring techniques [8, 9, 10, 11, 12] fail to satisfy this requirement, as they rely on *untrustworthy input.* These monitoring techniques exclusively rely on data structures maintained by software inside a target VM to derive views of the VM itself. It has been demonstrated that if the guest software is compromised, those data structures can be manipulated by attackers to circumvent such out-of-VM monitors [16, 17].

### 1.3.3 Inflexible Monitor Placement

Target systems and attacks are both moving targets. For example, the target system can be reconfigured or updated, or a new vulnerability or bug can be discovered. Many of these events require a corresponding update in the monitoring system. In addition, attacks are often carried out in multiple stages [18], with each stage requiring a different set of monitors to fully cover the trace of the attack.

For these reasons, monitoring systems need to be made ready for changes. Moreover, changes in a monitoring system should not be a source of downtime to target systems. This is however not the case for existing VM monitoring techniques, which require monitoring setup and configuration as a part of the target system boot process.

### 1.3.4 Incompatible Reliability and Security Monitoring

Reliability and security tend to be treated separately because they appear orthogonal: reliability focuses on accidental failures, security on intentional attacks. Because of the apparent dissimilarity between the two, tools to detect and recover from the different classes of failures and attacks are usually designed and implemented differently. So, integrating support for reliability and security in a single framework is a significant challenge.

Current VM monitoring techniques are no exception. While there is a substantial body of VM monitoring research dedicated to security monitoring, and some work dedicated to reliability, we are not aware of any previous effort toward combining these two subjects of monitoring.

The above four identified issues in VM monitoring hinder its adoption in production systems. Our research aims at (i) raising the awareness of those problems via demonstrations of real attacks and failures, and (ii) exploring new monitoring paradigms and methods that can resolve those problems.

## 1.4 Contributions

Based on the identification and demonstration of the limitations of existing VM monitoring, we propose three new continuous monitoring methods that

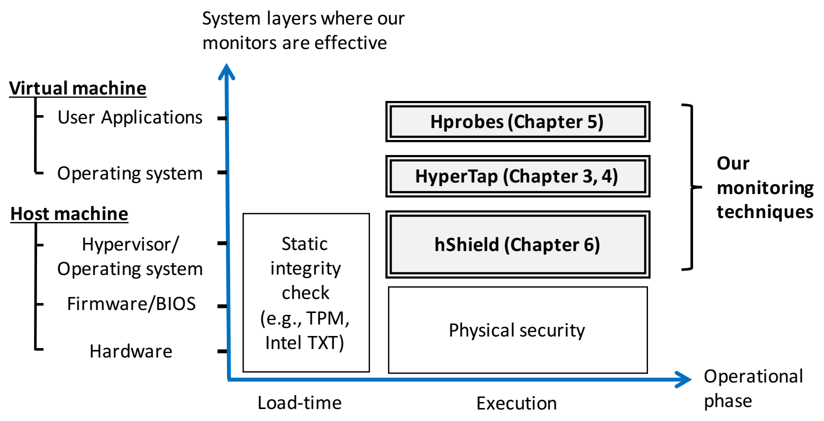


Figure 1.2: An illustration of our techniques to monitor a virtualized system at runtime (e.g., during execution). The y-axis represents the system layers from hardware at the bottom to user applications in a virtual machine (VM) at the top. The techniques are positioned at the layers where they provide monitoring: Hprobe monitors the VM's user applications, HyperTap monitors the VM's operating system, and hShield monitors the hypervisor.

address both VM attack and hypervisor attack scenarios mentioned in Section 1.2. Figure 1.2 summarizes our contributions organized in relation to the layers in the target system (y-axis) and system operational phase (x-axis).

### 1.4.1 Continuous Monitoring of Guest OS and Applications

For monitoring software running inside VMs, we introduce HyperTap and Hprobes, which are out-of-VM monitoring frameworks that facilitate detection of security and reliability incidents occurring inside a VM. These two frameworks can work in tandem to provide desirable monitoring features. HyperTap primarily focuses on monitoring the guest OS, while Hprobes adds guest application monitoring capability. On the one hand, HyperTap relies on fixed and well-defined hardware invariants to achieve robust and strong isolation with target VMs; on the other hand, Hprobes provides a mechanism for dynamic and flexible deployment of monitoring in the target VMs.

Both HyperTap and Hprobes employ the *event-driven monitoring* paradigm, which allows monitors to reactively respond to events of interest. In contrast to polling-and-scanning, event-driven monitoring exposes no temporal gap for failures and attacks to exploit. In addition, the event-driven monitoring mechanisms employed by these frameworks can capture target VMs' operational activities at various granularities, e.g., system call invocations and process/task-switching events. This provides a basis of support for a broad range of security and reliability enforcement policies.

To demonstrate the capabilities of HyperTap and Hprobes in supporting security and reliability monitoring, we introduced a set of low-cost and high-coverage monitors:

**HyperTap GOSHD – Guest OS Hang Detection.** GOSHD detected 99.8 percent of injected hang failures in a guest OS. GOSHD is also able to identify partial hangs, a new failure mode in multi-processor systems.

**HyperTap HRKD – Hidden-Rootkit Detection.** Rootkits are malicious computer programs that hide other programs from system administrators and security-monitoring tools. HRKD guarantees discovery of hidden processes and threads regardless of their hiding techniques. We verify the claim by testing HRKD against nine real-world rootkits in both Linux and Windows environments, with various types of hiding mechanisms.

**HyperTap PED – Privilege Escalation Detection.** In a privilege escalation attack, a process gains higher privileges than originally assigned to it in order to obtain unauthorized access to system resources. We demonstrate that PED can detect this class of attacks, including attacks that successfully bypassed Ninja [19], a real-world monitor, by exploiting temporal gaps created by polling-and-scanning monitoring.

**Hprobes EED – Emergency Exploit Detectors.** Often, a security vulnerability is discovered. After the vulnerability is made public, a patch takes time to be developed and must be put through a QA cycle. During this time, the target system is at risk of being attacked at the known vulnerability. We show that Hprobes can solve this practical problem by developing EED, a class of detectors that can prevent the exploitation of newly discovered vulnerabilities without patching the target system.

**Hprobes AHD – Application Heartbeat Detector.** One of the most basic reliability techniques used to monitor computing system liveness is a heartbeat detector. Using Hprobes, we constructed AHD, a monitor that directly measures the application's execution. That is, since probes are triggered by the application execution itself, they can be viewed as a mechanism for direct validation that the application is functioning correctly.

**Hprobes ILD – Infinite Loop Detector.** Infinite loops are a common failure that can cause process hangs. We demonstrated ILD, a monitor that uses Hprobes dynamic hook placement mechanism to measure the worst case execution time (WCET) [20] of a loop. The measure WCET is used to effectively detect infinite loops.

### 1.4.2 Continuous Monitoring of Hypervisor

HyperTap and Hprobes rely on the trustworthy of the underlining hypervisor to deploy their monitoring mechanisms. We demonstrate that this assumption can be violated by VM-escape attacks, which are attacks that compromise hypervisor executions via VM-exits, the VM-hypervisor interface provided by HAV (see Chapter 2 for a review of HAV). Based on the analysis of this threat model, we introduce hShield, which implements a novel Control-Flow Integrity (CFI) enforcement method to detect VM-escape attacks.

hShield continuously measures the CFI of every VM-exit handler, the basic block of hypervisor execution that handles VMs' privilege operations. The measurement is compared against a preconstructed Control-Flow Graph (CFG) to validate whether a valid path is executed. In hShield, a CFG is constructed using dynamic analysis, as opposed to the static analysis used by state-of-the-art techniques, to enhance the precision. We show that attacks can exploit the approximation of static analysis in building CFG to execute insecure paths, while our precise CFG cannot be exploited in this way.

In addition to demonstrating the strength of the constructed CFG, we show that our prototype of hShield is able to detect attacks crafted using a high-profile vulnerability in QEMU [21].

# CHAPTER 2

# BACKGROUND

*This chapter reviews cloud computing, basic concepts of virtualization, and state-of-the-art virtual machine monitoring research.*

## 2.1 Cloud Computing

Our search has been influenced by the progressive growth in adoption of the cloud computing model. This section briefly describes the context of this driving force.

Cloud computing is the most recent form of *utility computing*. In the sixties, computer scientist John McCarthy envisioned that computation could be delivered "as a public utility just as the telephone system is a public utility..." [22]. Here, public utility refers to services that rely on shared infrastructures rather than on local ones. A public utility can be consumed by anyone through subscription or pay-per-use. In the utility computing context, computer servers, disk storage, and network media are shared among users, who are billed per connecting time, CPU time, or used storage volume. These core principles of utility computing remain unchanged through time-sharing services in mainframes, server-hosting services, and now cloud computing.

After less than a decade of development, cloud computing has gone through several stages of adoption, from interest and exploration, to experimentation, and now to main production. In 2006, Amazon marked the beginning of cloud computing with the introduction of the Elastic Compute Cloud (EC2). In 2013, nearly 60 percent of small-to-medium businesses were using cloud computing services [23]. Also in 2013, the public cloud services market reached \$58 billion in revenues, and this market is projected to be close to \$200 billion by 2020 [24]. The growth in adoption, maturity, and financial viability is evident in that cloud computing has become a mainstream IT paradigm.

Cloud computing is a combination of three elements: *enablers*, *providers*, and *consumers*. *Enablers* are the key technologies that facilitate cloud computing. They include virtualization technologies for commodity hardware, high-speed networking, and distributed computing resource management. *Providers* are companies that successfully implemented the "As-a-Service" model to offer their infrastructure, Platform, and software to customers through subscription or pay-per-use. The early cloud providers include Rackspace, Amazon, Google, Microsoft, and IBM. Lastly, *consumers* are individuals and organizations who utilize cloud computing capacity to outsource their IT needs or to build their own applications and services. Cloud consumers wish neither to make an initial investment nor to continuously maintain a computing infrastructure below their own software stack.

The relationship between a provider and its consumers determines whether a cloud is *public* or *private*. A *public cloud* follows the standard public utility model, in which the provider delivers computing resources via the Internet to any consumers who agree to their terms, such as price and Service Level Agreement (SLA). A *private cloud* is one in which providers and consumers are strictly within the same organization. What makes a cloud private is the clear boundary of responsibility between the provider and its consumers. These two forms of cloud computing have been evolving in parallel to meet different customer requirements, such as legal compliance or security.

Cloud computing has some unique features. Cloud computing, like any other computing, specializes in processing and storing information. Unlike other utilities such as electricity, gas, or water, information itself does not have a physical form. As a result, information is not consumed in its use. It can be shared and used simultaneously by many parties without any loss to anyone. In addition, information is valueless without a context, e.g., it must be delivered in a timely way and to relevant destinations. These properties pose challenges when it comes to storing and processing information in a shared infrastructure. For example, How can we control information so that it flows only to intended destinations? Or, How can we guarantee the timeliness and completeness of information delivery in an infrastructure we do not have control over?

Cloud computing in its many forms has become a key computing infrastructure that supports business and governmental agencies across the globe.

This requires a guaranteed level of security and reliability in the infrastructure itself and in its services.

## 2.2 Overview of Machine Virtualization

*Machine virtualization* refers to the act of emulating the execution of a logical machine, called virtual machine (VM), in an environment controlled by software. The controlling software is called *Hypervisor* or *Virtual Machine Monitor* (VMM). In many cases, the term *virtualization* is used to indicate machine virtualization. However, virtualization in its broadest sense can include storage virtualization and network virtualization. Also, *server virtualization* and *desktop virtualization* are popularly used to indicate the particular types of machine, server or desktop computer, being virtualized.

Machine virtualization was pioneered by IBM back in the 1960s. The IBM S/360-67 computers were designed to expose virtual resources to allow multiple users to work on the same hardware. In 1974, Popek and Goldberg formalized the *trap-and-emulate* model of virtualization [25]. *Trapping* prevents the VM from taking privileged control, and *emulating* ensures that the semantics of the control do not violate the VM's expectations.

Trap-and-emulate can be done either (i) entirely in software via *binary translation* and/or *para-virtualization* or (ii) using *hardware-assisted virtualization* (HAV). Binary translation is an extreme case of this model, in which all instructions of the guest system are *trapped* and *emulated* by the host software. One advantage of this type of virtualization is that it allows running guest systems whose instruction sets are different from the instruction set supported by the physical processor. The down side of this technique, however, is the high performance overhead for virtualization. Typical examples of binary translation are QEMU and JVM. In para-virtualization, the *trapping* is done implicitly by directly modifying the guest system to notify the hypervisor to emulate the necessary requests. This modification reduces the run-time trapping overhead for virtualization. However, significant effort is often required to port the existing software system to para-virtualized environments.

HAV supports an unmodified guest OS with small performance overhead and significantly simplifies the implementation of hypervisors. Commodity

HAV is designed for x86 platforms include Intel VT-x [4] and AMD-V [5]. Nowadays, all commodity Intel and AMD processors are shipped with HAV support. The next section describes the principles of HAV in detail.

## 2.3 Hardware-Assisted Virtualization

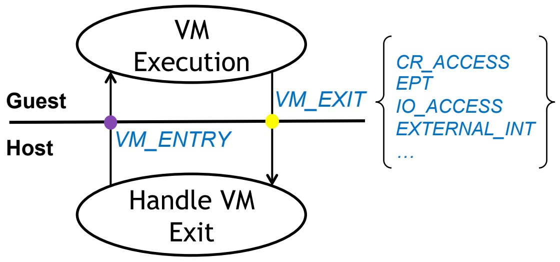


Figure 2.1: VM Exit mechanism in hardware-assisted virtualization.

In addition to x86's privilege rings, HAV defines guest mode and host mode execution. Certain operations (e.g., privileged instructions) are restricted in guest mode. If a guest attempts to execute a restricted operation, the processor relinquishes control to the hypervisor. If that happens, the processor fires a *VM Exit* event and transitions from guest mode to host mode. After the host has finished handling the exception, it resumes guest execution via a *VM Entry* event. Figure 2.1 illustrates this mechanism.

Table 2.1: Example of VM exit event types

| Event types | Description |
|---|---|
| EXCEPTION | An exception or interrupt is about to be delivered to guest VM |
| EXTERNAL_ INT | An external interrupt is about to be delivered to guest VM |
| CPUID | CPUID instruction is being executed in guest VM |
| WRMSR | WRMSR (write to MSR) instruction is being executed in guest VM |
| RDMSR | RDMSR (read from MSR) instruction is being executed in guest VM |
| HLT | HLT instruction is being executed in guest VM |
| INVD | INVD instruction is being executed in guest VM |
| INVLPG | INVLPG instruction is being executed in guest VM |
| VMCALL | VMCALL (or hypercall) instruction is being executed in guest VM |
| VMLAUNCH | VMLAUNCH instruction is being executed in guest VM |
| VMRESUME | VMRESUME instruction is being executed in guest VM |
| VMOFF | VMOFF instruction is being executed in guest VM |
| VMON | VMON instruction is being executed in guest VM |
| CR_ACCESS | Guest VM is attempting to access Control Register, e.g. Mov to CR3 or Mov from CR3 |
| DR_ACCESS | Guest VM is attempting to access Debug Register |
| IO_INST | Guest VM is attempting to execute an IO instruction (e.g. ioread, iowrite) |
| APIC_ACCESS | Guest VM is attempting to access APIC |
| EPT_VIOLA-TION | Guest VM is violating Extended Page Table (EPT) permission. |

Each type of restricted operation triggers a different type of VM Exit event. For example, if the guest attempts to modify the contents of a control register (CR), the processor fires a `CR_ACCESS` VM Exit event. Table 2.1 shows a partial list of VM Exit event types. In addition to the event, control fields and the state of the suspended VM are saved into a data structure (*VMCS* in Intel VT-x and *VMCB* in AMD-V).

### 2.3.1 Extended Page Tables (EPT)

In addition to virtualizing the CPU, one also needs to be concerned with the virtualization of the memory management unit (MMU). Virtual memory is the cornerstone of process isolation in every modern OS, and therefore it is a necessary feature for VMs. Earlier implementations of x86 hypervisors used Shadow Page Tables, which are data structures that contain mappings from *guest virtual addresses* to *host physical addresses*. Shadow Page Tables use a costly VM Exit synchronization technique to match the shadow structures with the hardware page tables. To avoid this overhead, CPU vendors added a feature, *Second-Level Address Translation* (SLAT) or *Two-Dimensional Paging*, to their virtualization extensions. This technology is called Extended Page Tables (EPT) in the Intel Architecture and Nested Page Tables (NPT) in AMD. With EPT, the hardware uses a second set of page tables to translate from *guest physical addresses* to *host physical addresses*. Handling this translation in hardware eliminates most VM Exits used to synchronize guest page tables. Therefore, EPT provides performance benefits in most cases, but it results in very costly TLB misses because an additional set of page tables must be traversed [26]. See Chapter 28 of the *Intel Software Developer Manual* [4] for more details.

EPT also allows specification of access permissions for guest memory pages, namely 'read,' 'write,' and 'execute'. Guest attempts at unauthorized accesses cause `EPT_VIOLATION` VM Exits.

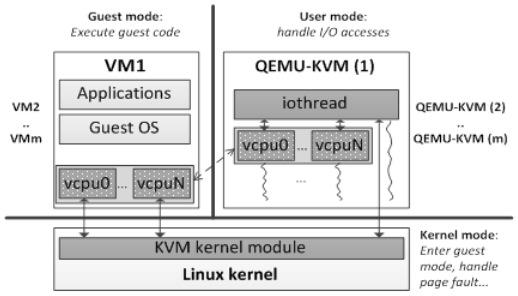


Figure 2.2: KVM architecture.

### 2.3.2 Examples of Open Source HAV-based Hypervisor

KVM

KVM (the Linux Virtual Machine Monitor) is a kernel extension, which, after loading, turns the Linux kernel into a virtual machine monitor or hypervisor.

Figure 2.2 depicts the KVM architecture. The hypervisor consists of a KVM kernel module and one qemu-kvm user process for each VM (or guest system). The KVM kernel module leverages the hardware virtualization provided by recent x86 processors (e.g., Intel VT and AMD-V) to emulate virtual CPUs (vcpu). This module is also responsible for entering guest mode and handling memory management of the VM. After entering guest mode, the guest code, including both the guest OS and the guest applications, is executed natively, rather than using emulation or binary translation, until it needs I/O access or receives incoming interrupts. In KVM architecture, all IO operations are forwarded to user mode, which is the hardware emulator qemu-kvm. A qemu-kvm handles all the I/O accesses of a VM. This is a multi-threaded process. It creates one thread for each vcpu and one thread to simulate other devices such as NIC (network interface card) controller and disk controller.

In a cloud environment, the providers often do not control the user's workload running in the guest VM. However, they need to take care of the software and hardware layers running below the guest OS. In this case, there are qemu-kvm processes, the KVM kernel module, the host OS, the management system, and the physical hardware. Ideally, a failure should not propagate (i) from guest mode to user mode or kernel mode, (ii) from user mode to kernel mode or (iii) from hypervisor to the management system.

XEN HVM

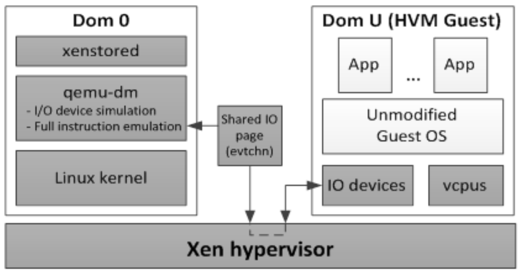


Figure 2.3: XEN architecture in hardware-assisted virtualization (HVM)

A XEN virtualization system is powered by the XEN hypervisor, which is the most privileged software layer operating right on top of the hardware. On top of the XEN hypervisor, one or more guest operating systems can be hosted. After the hypervisor boots, it automatically loads the first guest operating system (Dom0). Dom0 has special management privileges with respect to other VMs, called DomU. By default, Dom0 has direct access to the physical hardware.

To separate the mechanism and policy, XEN hypervisor exports a control interface to Dom0. Application-level management software running in Dom0 uses this interface to manage the system's resources. For example, xenstored is used to store information about the domains during their execution and to create and control domU devices. To support the unmodified guest OS, XEN also uses a customized version of qemu (qemu-dm) to simulate virtual hardware. Similarly to KVM, each VM is coupled with one qemu-dm process running in Dom0. In the context of this thesis, XEN hypervisor and the user-application management software are the targets for fault-injection-based analysis. Figure 2.3 depicts XEN architecture in hardware-assisted virtualization (HVM) mode.

## 2.4 Virtual-Machine-based Monitoring

Virtualization has enabled a set of out-of-VM security monitoring techniques [8, 9, 10, 11, 27, 13, 12, 28, 29] as well as new techniques to enhance the security of traditional in-VM security monitoring tools [30, 31].

*Out-of-VM monitoring* separates monitoring tools from protected VMs to achieve stronger malicious-attack resistance than conventional in-VM (or in-host in non-virtualized environments) monitoring techniques. Out-of-VM techniques can be further classified into *passive* and *active* monitoring.

*Passive out-of-VM monitoring* techniques, such as LiveWire [8], VMWatcher [9], XenAccess [10], Virtuoso [11], and OSck [12], use introspection to perform integrity checking by scanning or polling states of protected VMs. Since manual extracting of meaningful introspection from a guest OS is a costly process, recent studies have proposed methods to semi-automate [11] or fully automate [27] the generation of VMI tools. Even though the burden of VMI tool development can be offloaded to the complexity of the automation tools, the runtime performance overhead of introspection is still high. Furthermore, many intrusion detection techniques and failure detection techniques require intercepting and responding to events generated by protected VMs [13], which this underlining introspection technique cannot do.

*Active out-of-VM monitoring* has been proposed to address the above limitations. Lares [13], for example, is an architecture to securely place hooks in protected VMs (or untrusted VM in Lares terminology) to intercept their events and then transfer these events to security applications running in trusted VMs. Lares, however, experiences two drawbacks: (i) high performance overhead due to the implementation of memory protection at byte-level (to protect hooks) and frequent world-switching (to transfer event from untrusted VM to trusted VM) and (ii) vulnerability to event manipulation because it cannot guarantee the integrity of the sources that generate the events. In addition, Lares' hook placement mechanism is intrusive to the guest system. To reduce the amount of manual intervention in the process, the authors of [32] propose a method to automatically identify locations to place useful application-aware hooks.

Although out-of-VM monitoring has raised the bar for security applications, the underlying principle of using guest OS invariants to detect security violations has several caveats. First, due to the complexity of modern full-fledged OSs, it is impossible to cover all the invariants that could lead to security violations. In reality, monitored OS invariants often belong to known attacks [37]. Second, despite the physical isolation with protected VMs, these types of monitoring tools still use internal guest OS states as input driving their decisions. As demonstrated by [16], by using DKSM (Di-

Table 2.2: Comparison of VM-based monitoring techniques

| | LiveWire[8] | Lares[13]/SIM[30] | XenAccess[10] | VMWatcher[9] | Antfarm[28]/Lycosid[29] | Ether[33]/Nitro[34] | OSck [12] | Virtuoso[11] | VMST[27] | TxIntro[35] | **HyperTap**[14] | **Hprobe**[36] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OS/HW invariants | OS | OS | OS | OS | HW | HW | OS | OS | OS | OS | **HW** | **OS** |
| Passive/Active Mon. | P | A | P | P | P | P | P | P | P | P | **A** | **A** |
| Changes to VM | N | Y | N | N | N | N | N | N | N | N | **N** | **N** |
| Changes to Hypervisor | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | **Y** | **Y** |
| Changes to HW | N | N | N | N | N | N | N | N | N | N | **N** | **N** |
| Custom Auditors | Y | Y | Y | Y | Y | N | N | Y | Y | Y | **Y** | **Y** |
| Userspace Monitoring | Y | Y | Y | Y | Y | N | N | Y | Y | Y | **N** | **Y** |
| Online Monitoring | Y | Y | Y | Y | Y | Y | N | Y | Y | N | **Y** | **Y** |
| Auto-generate Invariants | N | N | N | N | N | N | N | Y | Y | Y | **N** | **N** |

rect Kernel Structure Manipulation) technique, an attacker inside the VM can control the view of an outside VMI tool. Hund et al. [17] present a technique, based on return-to-libc attacks [38], to implement rootkits that can bypass kernel code integrity checking, which makes these rootkits even stealthier under the view of VMI. HyperTap-based VM monitoring relies primarily on the hardware architectural invariants, which directly reflect end behaviors of protected VMs. Therefore our monitors can detect the abnormal behaviors (e.g., hiding a process) regardless of their mechanisms (e.g., corrupting one of the guest OSs invariants), as long as the hypervisor is not compromised.

Previous studies [28, 29, 33, 39] show how the hardware architectural state can be used to interpret a guest OS's operations. For example, Antfarm [28] and its extension Lycosid [29] describe a guest user process counting technique based on monitoring virtual memory (i.e., tracking `CR3` in x86). Ether [33] utilizes the VM Exit mechanism provided by HAV to record traces of guest VM execution for offline malware analysis. HyperTap builds on those concepts to provide robust online monitoring for both reliability and security.

*Out-of-VM failure detection* has also been studied in previous research. Virtualization has been used to enhance the reliability of the overall software system. In addition, previous studies have explored out-of-VM failure detection. Pelleg et al. [40] used decision tree, a supervised machine learning technique, to detect failure in VMs using six hypervisor-level counters. This technique, however, requires training datasets from previously known failures of the monitored system.

Table 2.2 compiles a comparison of existing VM monitoring frameworks, including HyperTap [14] and Hprobe [36].

# CHAPTER 3

# HYPERTAP: VIRTUAL MACHINE MONITORING USING HARDWARE ARCHITECTURAL INVARIANTS

*This chapter presents HyperTap, a hypervisor-level framework that efficiently supports continuous monitoring of VMs' operating system. The core principle of HyperTap is to rely on hardware architectural invariants to provide a low-cost, reliable, and hard-to-bypass event interception mechanism.*

## 3.1 Introduction

Reliability and security (RnS) are two essential aspects of modern highly connected computing systems. Traditionally, reliability and security tend to be treated separately because of their orthogonal nature: while reliability deals with accidental failures, security copes with intentional attacks against a system. As a result, mechanisms/algorithms addressing the two problems are designed independently, and it is difficult to integrate them under a common monitoring framework.

Addressing RnS aspects separately may lead to unforeseen consequences. For example, a reliability monitor (e.g., a heartbeat server) may have a vulnerability that allows remote attackers to exploit the system. On the other hand, a security monitor may introduce a new failure mode that the current system is not designed to handle. Furthermore, different modules' design and implementation may not be compatible. For instance, suppose two monitors both require exclusive access to a resource, e.g., a performance register. Such monitors cannot co-exist in the same system. This situation places system designers in a difficult position, in which they must trade off one essential quality for another. In a milder scenario, the system has to pay a combinational cost, e.g., development, deployment, and runtime performance costs, of both solutions.

In this chapter, we identify the commonalities between reliability and security monitoring to guide the development of suitable frameworks for combining both uses of monitoring. We apply our observations in the design and implementation of the HyperTap framework for virtualization environments.

A monitoring process can be divided into two tightly coupled phases: *logging* and *auditing* [41]. In the *logging* phase, relevant system events (e.g., a system call) and states (e.g., system call parameters) are captured. In the *auditing* phase, these events and states are analyzed, based on a set of policies that classify the state of the system, e.g., normal or faulty. Based on that model, we observe that although RnS monitors may apply different policies during the auditing phase, they can utilize the same event- and state- logging capability. This observation suggests that the logging phases of multiple RnS monitors need to be combined into a common framework. Unification of logging phases brings further benefits; namely, it avoids potential conflict between different monitors that track the same event or state, and reduces the overall performance overhead of monitoring.

A unified logging framework for RnS must be founded on an *isolated root of trust* and have support for *active monitoring.* Current virtual machine monitoring techniques, e.g., Virtual Machine Introspection (VMI), either exhibit neither of those two properties, or offer only one at time. An isolated root of trust asserts that the source of captured events and state cannot be tampered with by actors inside target systems. Traditional VMI techniques fail on that condition, as they choose to rely solely on the guest operating system (OS) to report its own state. An example of that violation is presented in [16] (the issue is further discussed in Section 3.2.2). For RnS monitoring, active monitoring (or *event-driven monitoring*) has been shown to be more advantageous than passive monitoring (or *state polling*), as the former can capture operational events in addition to the system's state [13]. Furthermore, active monitoring can overcome the time sensitivity of passive monitoring; e.g., it can detect short latency failures and transient attacks [42], as further illustrated in Section 3.2.3.

In order to fulfill the requirements stated above, we present a framework implemented at the hypervisor level called *HyperTap*, that provides an event logging infrastructure suitable for implementing various types of RnS policies for virtual machines (VMs). In HyperTap, the logging phase is common for all monitors and constitutes the core of the framework. The auditing phase of each monitor is implemented and operated independently. To achieve an isolated root of trust, HyperTap employs hardware architectural invariants, which cannot be modified by attackers and failures inside VMs. These invariants hold under assumptions about the trustworthiness of the hypervisor and hardware stated in Section 3.4.1. In order to support active monitoring and intercept a wide range of system events, HyperTap utilizes the hardware assisted virtualization (HAV) event generation mechanism. The events are then delivered to registered auditors which realize a variety of RnS monitoring policies.

In order to demonstrate the feasibility of HyperTap as a framework that unifies RnS monitoring for virtualized environments, we describe the design and evaluation of three practical lightweight auditors: Guest Operating System Hang Detection (GOSHD), Hidden Rootkit Detection (HRKD), and Privilege Escalation Detection (PED). The GOSHD and HRKD auditors are chosen to show that a common event, e.g., context switching, can be simultaneously used for both reliability and security monitoring. The PED auditor is chosen to show the advantages of active monitoring over passive monitoring. In addition to facilitating the unification of RnS monitors, HyperTap's dependable hardware invariants and active monitoring mechanism enable auditors with high detection coverage. GOSHD can detect 99.8% of injected hang failures, including partial hang failures in multiprocessor VMs – a new failure mode revealed by GOSHD. HRKD can detect both hidden processes and kernel threads regardless of their hiding mechanisms. And PED can detect all four types of proposed attacks that defeat Ninja [19], a real-world privilege escalation detector that uses passive monitoring.

## 3.2 Monitoring Principles

This section discusses the benefits of (i) having a unified logging channel for all monitors, (ii) using active monitoring instead of passive monitoring, and (iii) placing the root of trust at hardware invariants for virtual machine monitoring.

### 3.2.1 Unified Logging

It is not uncommon for co-deployed logging mechanisms to conflict. For instance, two monitors relying on a certain counter that only allows exclusive access cannot use it simultaneously. A concrete example would be to deploy both the failure detection technique proposed in [43] and the malware detection technique proposed in [44] in the same system, as they both use hardware performance counters. In addition, one monitor may become a source of noise for other monitors. For example, intrusive logging could generate an excessive number of events.

The problem can be solved by unifying logging for co-located monitors. Unified logging is responsible for (i) retrieving common target system events and states, and then (ii) streaming them in a timely manner to customizable auditors, which enforce RnS policies.

Aside from avoiding potential conflicts, the combination of logging phases yields additional benefits. It can reduce the overall performance overhead of combined monitors. To ensure the consistency of captured states and events, logging is often a blocking operation. Once the event and state have been logged, an audit can be performed in parallel with execution of the target system. Therefore, combining blocking logging phases boosts performance, even in cases where the captured states differ. Furthermore, this approach inherits other benefits of the well-known divide-and-conquer strategy: it allows one to focus on hardening the core logging engine, and enables incremental development and deployment of auditing policies.

### 3.2.2 Achieving Isolation via Architectural Invariants

An *OS invariant* is a property defined and enforced by the design and implementation of a specific OS, so that the software stack above it, e.g., user programs and device drivers, can operate correctly. In the context of VMI, OS invariants allow the internal state of a VM to be monitored from the outside by decoding the VM's memory [9, 10, 11, 8, 12]. No user inside a VM can interfere with the execution of outside monitoring tools. However, monitoring tools still share input, e.g., a VMs' memory, with the other software inside VMs. Therefore, those monitoring tools are vulnerable to attacks at the guest system level, as demonstrated in [16, 17, 45].

An *architectural invariant* is a property defined and enforced by the hardware architecture, so that the entire software stack, e.g., hypervisors, OSes, and user applications, can operate correctly. For example, the x86 architecture requires that the `CR3` and `TR` registers always point to the running process's Page Directory Base Address (`PDBA`) and Task State Segment (`TSS`), respectively. Hardware invariants and HAV features have been studied in the context of security monitoring [28] and offline malware analysis [33].

We find that architectural invariants, particularly the ones defined by HAV, provide an outside view with desirable features for VM RnS monitoring. The behaviors enforced by HAV involve primitive building blocks of essential OS operations, such as context switches, privilege level (or ring) transfers, and interrupt delivery. Furthermore, strong isolation between VMs and the physical hardware ensures the integrity of architectural invariants against attacks inside VMs. Software inside VMs cannot tamper with the hardware as it can with the OS. In this study, we explore the full potential of HAV for online enforcement of RnS policies.

However, relying solely on architectural invariants and ignoring OS invariants would widen the semantic gap separating the target VM and the hypervisor. The reason is that many OS concepts, such as user management (e.g., processes owned by different users), are not defined at the architectural level. In this study, we propose to *use architectural invariants as the root of trust when deriving OS state.* For example, the `thread_info` data structure in the Linux kernel containing thread-level information can be derived from the `TSS` data structure, a data structure defined by the x86 architecture.

In order to circumvent our OS state derivation, an attack would need to change the layout of OS-defined data structures (e.g., by adding fields to an existing structure that point to tainted data). Changing data structure layout, as opposed to changing values, is difficult for attackers, because (i) it involves significant changes to the kernel code that references the altered fields, and (ii) it would need to relocate all relevant kernel data objects. Not only are those attacks difficult to perform on-the-fly, but since malware always tries to minimize its footprint, our approach significantly impedes would-be attackers.

### 3.2.3 Robust Active Monitoring

Passive monitoring is suitable for persistent failures and attacks, because it assumes the corrupted or compromised state remains in the system sufficiently longer than the polling interval. That assumption does not hold in many RnS problems. For example, the majority of crash and hang failures in Linux systems have short failure latencies (the time for faults to manifest into failures) [46]. An unnecessarily long detection latency, e.g., caused by polling monitoring, would result in subsequent failure propagation or inefficient recovery (e.g., multiple roll-backs).

As we demonstrate in Section 4.3.2, a transient attack can be combined with other techniques to create a stealthy attack that can defeat passive monitoring.

*Active monitoring*, on the other hand, possesses many attractive features. Since it is event-driven, there is no time dependence that can be exploited. Furthermore, active monitoring can capture system activities in addition to the system state, which passive monitoring provides. System activities are the operations that transition a system from one state to another. Invoking a system call is an example of a system activity. In many cases, information about system activities is crucial to enforcing RnS policies.

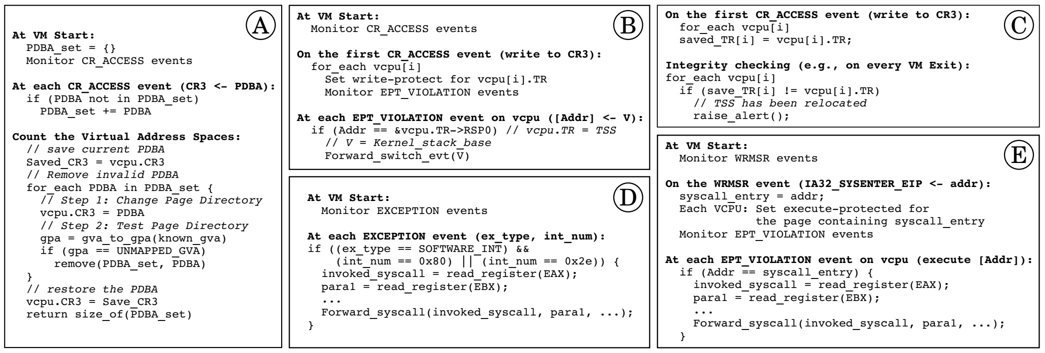
Active monitoring is not foolproof, as it can suffer from *event bypass* attacks. If an attack can prevent or avoid generation of events that trigger logging, it can bypass the monitor. To make active monitoring robust, we propose to use hardware invariants, specifically the VM Exit feature provided by HAV, to generate events. Section 3.3 presents the hardware invariants used to ensure the trustworthiness of generated events.

## 3.3 Hardware Invariants for VM Logging

This section describes events that can be monitored via hardware invariants and VM Exit events, the core mechanism of HyperTap's shared logging channel. Table 3.1 summarizes guest systems' internal operations, the hardware invariants, and the types of VM Exit events associated with them. The following sub-sections detail the use of these invariants.

Table 3.1: Summary of guest internal events and related VM Exit types

| Monitoring Category | Guest event | Related VM Exit | Architectural Invariant |
|---|---|---|---|
| **Context switch interception (§3.3.1)** | Process context switch (§3.3.1) | `CR_ACCESS` | The `CR3` register always points to the `PDBA` of the running process<br>Writes to CR registers cause `CR_ACCESS` VM Exits |
| | Thread switch (§3.3.1) | `EPT_VIOLATION` | The `TR` register always points to the `TSS` structure of the running process<br>`TSS.RSP0` is unique for each thread |
| **System call interception (§3.3.2)** | Interrupt-based system call (§3.3.2) | `EXCEPTION` | Software interrupts cause `EXCEPTION` VM Exits |
| | Fast system call (§3.3.2) | `WRMSR`,<br>`EPT_VIOLATION` | `SYSENTER`'s target instruction is stored in an MSR register<br>Write to MSR registers causes `WRMSR` VM Exit |
| **I/O access interception (§3.3.3)** | Programmed I/O | `IO_INST` | Execution of I/O instructions (e.g., `IN, INS, OUT, OUTS`) |
| | Memory mapped I/O | `EPT_VIOLATION` | Access to memory mapped I/O areas, which are set as protected |
| | Hardware interrupt | `EXTERNAL_INT` | Hardware interrupt delivery causes `EXTERNAL_INT` VM Exits |
| | I/O APIC access | `APIC_ACCESS` | I/O Advance Programmable Interrupt Controller (APIC) events |
| **Low-level interception (§3.3.4)** | Memory access | `EPT_VIOLATION` | Accesses to memory regions with proper permissions cause `EPT_VIOLATION` VM Exits |
| | Instruction execution | `EPT_VIOLATION` | Execution of instructions from non-executable regions causes `EPT_VIOLATION` VM Exits |



```
At VM Start:
  PDBA_set = {}
  Monitor CR_ACCESS events

At each CR_ACCESS event (CR3 <- PDBA):
  if (PDBA not in PDBA_set)
    PDBA_set += PDBA

Count the Virtual Address Spaces:
  // save current PDBA
  Saved_CR3 = vcpu.CR3
  // Remove invalid PDBA
  for_each PDBA in PDBA_set {
    // Step 1: Change Page Directory
    vcpu.CR3 = PDBA
    // Step 2: Test Page Directory
    gpa = gva_to_gpa(known_gva)
    if (gpa == UNMAPPED_GVA)
      remove(PDBA_set, PDBA)
  }
  // restore the PDBA
  vcpu.CR3 = Save_CR3
  return size_of(PDBA_set)
```

```
At VM Start:
  Monitor CR_ACCESS events

On the first CR_ACCESS event (write to CR3):
  for_each vcpu[i]
    Set write-protect for vcpu[i].TR
    Monitor EPT_VIOLATION events

At each EPT_VIOLATION event on vcpu ([Addr] <- V):
  if (Addr == &vcpu.TR->RSP0) // vcpu.TR = TSS
    // V = Kernel_stack_base
    Forward_switch_evt(V)
```

```
On the first CR_ACCESS event (write to CR3):
  for_each vcpu[i]
  saved_TR[i] = vcpu[i].TR;

Integrity checking (e.g., on every VM Exit):
for_each vcpu[i]
  if (save_TR[i] != vcpu[i].TR)
    // TSS has been relocated
    raise_alert();
```

```
At VM Start:
  Monitor EXCEPTION events

At each EXCEPTION event (ex_type, int_num):
if ((ex_type == SOFTWARE_INT) &&
    (int_num == 0x80) || (int_num == 0x2e)) {
  invoked_syscall = read_register(EAX);
  para1 = read_register(EBX);
  ...
  Forward_syscall(invoked_syscall, para1, ...);
}
```

```
At VM Start:
  Monitor WRMSR events

On the WRMSR event (IA32_SYSENTER_EIP <- addr):
  syscall_entry = addr;
  Each VCPU: Set execute-protected for
             the page containing syscall_entry
  Monitor EPT_VIOLATION events

At each EPT_VIOLATION event on vcpu (execute [Addr]):
  if (Addr == syscall_entry) {
    invoked_syscall = read_register(EAX);
    para1 = read_register(EBX);
    ...
    Forward_syscall(invoked_syscall, para1, ...);
  }
```

Figure 3.1: Pseudo-code for each interception algorithm. (A): Process Counting Algorithm, (B): Thread switch interception, (C): TSS integrity checking, (D): Interrupt-based system call interception, (E): Fast system call interception

### 3.3.1 Context Switch Interception

#### Process Switch Interception

**Architectural Invariant**. Process switches can be observed by monitoring `CR_ACCESS` VM Exit events. In x86, the `CR3` register, or Page Directory Base Register (`PDBR`) contains the Page Directory Base Address (`PDBA)` for the virtual address space of the running process. As this base address is unique for each user process, we can use it as a process identifier.

**Process Counting Algorithm**. We can count the number of processes running on a guest VM by monitoring `CR_ACCESS` events. This algorithm is independent of any data structure the guest OS uses to manage its processes.

Figure 3.1A shows the pseudo-code for the process counting algorithm. The set of PDBAs (`PDBA_set`) is empty when the guest OS boots up. At each `CR_ACCESS` event in which `CR3` is modified (`CR3 <- PDBA`), the algorithm updates `PDBA_set` with the value that will be written to `CR3`.

Thread Switch Interception

Monitoring of thread[1] switches requires more effort than tracking `CR_ACCESS` events, as threads can share the same virtual address space. In addition, a thread can reuse the virtual address space of another process (e.g., Linux kernel threads).[2]

**Architectural Invariant**. In order to manage threads, the x86 processor uses the Task Register (`TR`) and Task-State Segment (`TSS`) structures. The `TSS`, stored in main memory, holds the stack pointers of a task for different privilege levels, and the `TR` points to the `TSS` structure of the current task. The `TSS` is also used to support privilege protection. Each time execution transfers from user level (3) to kernel level (0), the kernel stack pointer is automatically loaded from the `TSS` by the CPU (e.g., `RSP <- TSS→RSP0`). Since all kernel threads share the same virtual address range, each has a separate address range for its stack. Therefore, the kernel stack pointer (`RSP0`) stored in the `TSS` can be used as a thread identifier.

**Thread Switch Interception Algorithm**. Each thread switch modifies the `TSS` stored in memory. Therefore, we can track thread switches by setting memory access permissions. Specifically, on a guest system with EPT, a write to an EPT write-protected address triggers an `EPT_VIOLATION` VM Exit. We use this mechanism to track the kernel stack pointer.

Figure 3.1B shows the pseudo-code for this algorithm. After the guest OS finishes setting up its data structures (e.g., the `CR3` register gets written for the first time), the algorithm sets all pages that contain `TSS` structures (one per vCPU) as write-protected. Each time a `TSS` structure is modified, the hypervisor gets notified by an `EPT_VIOLATION` event.

[1]A thread is equivalent to a task in the x86 architecture.

[2]*kthreads* reuse the virtual address space of the previously scheduled process. All processes in Linux have the same kernel address range. Windows does not have standalone kernel threads.

### 3.3.2 System Call Interception

System calls allow user mode processes to invoke kernel mode functions. At the hardware level, a system call transfers the CPU from user to kernel mode. That transfer from a lower to higher privilege is strictly checked by the processor: it must be done through pre-defined *gates*. This section describes techniques to intercept two types of system calls: *interrupt-based system calls* and *fast system calls*.

#### Interrupt-based System Calls

The legacy method for issuing a system call in x86 is to raise a software interrupt. For example, Linux uses `INT $0x80` and Windows uses `INT $0x2E` to issue system calls. The interrupt handler routine is the common gate for all system calls, and parameters of system calls are passed through general-purpose registers.

**Architectural Invariant**. In a VM, each software interrupt triggers an `EXCEPTION` VM Exit.[3]

**Interrupt-based System Call Interception Algorithm**. We developed an algorithm that intercepts interrupt-based system calls, shown in Figure 3.1D. If the type and number of the interrupt indicate a system call, the algorithm records all the registers that could carry the system call's parameters and then generates a notification regarding the system call.

#### Fast System Calls

A fast system call mechanism was added to x86 with the `SYSENTER/SYSEXIT` instruction pair for Intel processors and the `SYSCALL/SYSRET` instructions for AMD processors.

**Architectural Invariant**. The `SYSENTER` instruction takes input from Model Specific Registers (MSRs) and general-purpose registers. For example, `SYSENTER`'s target instruction address is stored in the `IA32_SYSENTER_EIP` MSR. An MSR can only be modified via a `WRMSR` instruction, a privileged instruction that causes `WRMSR` VM Exits.

[3]Intel VT-x allows selection of which interrupts cause `EXCEPTION` VM Exits via an `EXCEPTION_BITMAP`.

**Fast system call interception algorithm**. Figure 3.1E contains pseudocode for fast system call interception. The algorithm uses `WRMSR` events to identify the address of the system call entry point in the guest VM. The address is set to execute-protect so that a guest's attempt to execute the system call entry point will generate an `EPT_VIOLATION` VM Exit.

### 3.3.3 I/O Access Interception

A primary function of the hypervisor is to multiplex I/O devices for its VMs, except when a VM is given exclusive access via an I/O pass-through mode. HAV provides several VM Exits that the hypervisor can use to capture IO accesses from guest VMs. We categorize I/O accesses into three types:

**Programmed I/O (PIO)** is performed through I/O instructions, such as `IN` and `OUT`. These instructions trigger `IO_ACCESS` events when executed in guest mode.

**Memory Mapped I/O (MMIO)** is performed through instructions that manipulate memory (e.g., `MOV`, `AND`, `OR`). In order to trap MMIO, the hypervisor sets memory protection for the allocated MMIO area so that accesses to this area will trigger `EPT_VIOLATION` events.

**I/O interrupts** are interrupts raised by physical devices to notify guest VM about I/O-related events (e.g., an incoming network packet). The presence of a pending interrupt causes either an `EXTERNAL_INT` or `APIC_ACCESS` VM Exit event.

Because of the diversity of I/O devices, details for each type of device are not covered, and it is up to implementers to choose an appropriate mechanism.

### 3.3.4 Fine-grained Interception

The EPT feature presented in Section 2.3.1 makes it possible to track a guest's execution at the single instruction and memory access level by setting appropriate access permissions. However, that fine-grained interception incurs a significant performance cost. To minimize its impact, an auditor should make use of that feature only for selective critical protection.

## 3.4 Framework and Implementation

Following the principles presented in the previous section, here we describe the design and implementation of HyperTap with KVM, as illustrated in Figure 3.2.

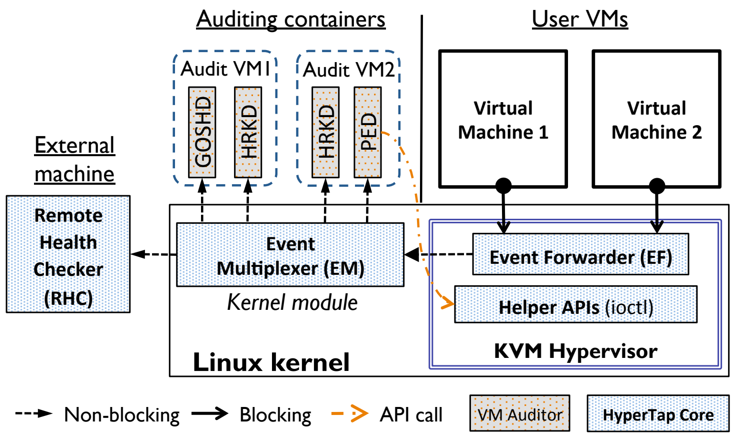


Figure 3.2: Implementation of HyperTap in the KVM hypervisor. The hypervisor is modified to forward VM Exit events to the Event Multiplexer (EM), which is implemented as a separate kernel module. The EM forwards events to registered auditors running as user processes inside auditing containers. The Remote Health Checker (RHC) monitors the hypervisor's liveness.

### 3.4.1 Scope and Assumptions

HyperTap integrates with existing hypervisors to safeguard VMs against failures and attacks. It aims to make this protection transparent to VMs by utilizing existing hardware features. Thus, HyperTap does not require modification of either the existing hardware or the guest OS's software stack.

HyperTap's implementation assumes that the underlying hardware and hypervisor are trusted. Although extra validation and protection for the hardware and hypervisor could address concerns about the robustness of different hypervisors against failures and attacks, these issues are beyond the scope of this work.

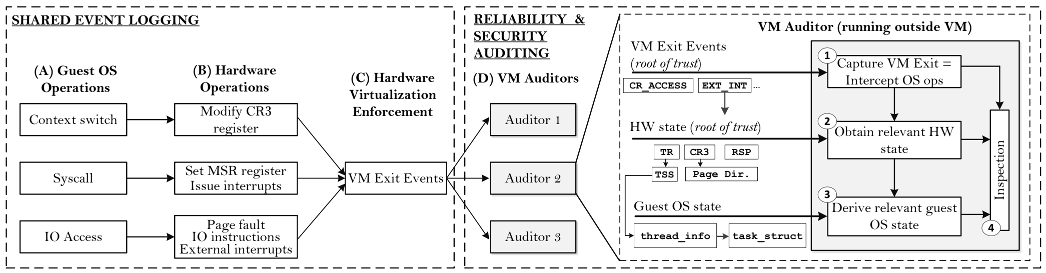


Figure 3.3: HyperTap monitoring framework: (A) Guest OS operations that are subjects of the monitors; (B) Hardware operations that are required to perform each guest OS operation; (C) VM Exit events that are generated before logged operations are performed; (D) The captured events are delivered to auditors running outside the VM.

### 3.4.2 Monitoring Workflow

Figure 3.3 depicts the overall workflow of HyperTap. The left side of the figure illustrates how the shared event logging mechanism works and the right side describes the auditing phase. HyperTap utilizes HAV to intercept the desired guest OS operations through VM Exit events generated by corresponding hardware operations. Since the HAV VM Exit mechanism is not designed to intercept all desired operations, e.g., system calls, Section 3.3 presents algorithms to generate VM Exit events for such operations.

HyperTap supports a wide range of events, from coarse-grained events, such as process context switches, to finer-grained events, such as system calls, and very fine grained events, such as instruction execution and memory accesses. That variable granularity ensures that HyperTap can be adopted for a broad range of RnS policies.

HyperTap delivers captured events to registered auditors, which implement specific RnS policies. An auditor starts by registering for a set of events needed to enforce its policy. Upon the arrival of each event, the auditor analyzes the state information associated with the event. Auditors are associated with VMs and each VM can have multiple auditors.

HyperTap also provides an interface that allows auditors to control to target VMs. For example, the auditing phase is non-blocking by default, but an auditor may pause its target VM during analysis in order to stop the VM during an attack, or roll-back the VM when it detects a non-recoverable failure.

### 3.4.3 Implementation

This subsection presents the integration of HyperTap with KVM [6], hypervisor built with HAV as a Linux kernel module. Figure 3.2 depicts the deployment of HyperTap's components.

HyperTap's unified logging channel is implemented through two components: an *Event Forwarder (EF)* and an *Event Multiplexer (EM)*. The EF is integrated into the KVM module, and forwards VM Exit events and relevant guest hardware state to the EM. By default, events are sent non-blocking to minimize overhead. The EM, which is implemented as another Linux kernel module in the host OS, buffers input events from the EF and delivers them to the appropriate auditors.

The EM is also responsible for sampling VM Exit events that are sent to a Remote Health Checker (RHC) running in a separate machine. The RHC server acts as a heartbeat server to measure the intervals between received events. If no events are received after a certain amount of time, it raises an alert about the liveness of the monitoring system.

Auditors are implemented as user processes inside auditing containers[4] running on the host OS. Compared to the dedicated auditing VM used in previous work [11] [13], this approach offers multiple benefits. First, it provides lightweight attack and failure isolation among different VMs' auditors, and between auditors and the host OS. Second, it simplifies implementation and reduces the performance overhead of event delivery from the EM module. Finally, it allows the integration of auditors into existing systems, since containers are robust and compatible with most current Linux distributions.

We needed to add less than 100 lines of code to KVM to implement the EF component and export Helper APIs.

## 3.5 Conclusion

This chapter presents principles for unifying RnS monitoring. We identify the boundary dividing the logging and auditing phases in monitoring processes. That boundary allows us to unify and develop dependable logging mechanisms. We demonstrate the need for an isolated root of trust and

[4]We use Linux containers (LXC) `http://linuxcontainers.org/`

active monitoring to support a wide variety of RnS monitors. We applied those principles when developing HyperTap, a framework that provides unified logging, based on hardware invariants, to safeguard VM environments. In the next chapter, one reliability monitor and two security monitors will be introduced and evaluated to demonstrate the feasibility and practicality of HyperTap.

# CHAPTER 4

# BUILDING DETECTORS USING HYPERTAP

*This chapter expands on the techniques presented in the previous chapter to demonstrate how to build auditors using HyperTap. Specifically, we present two examples that showcase how RnS monitoring can be combined (GOSHD and HRKD) and one example that demonstrates the effectiveness of active monitoring (PED).*

## 4.1 GOSHD – Guest OS Hang Detection

### 4.1.1 Failure Model

We consider an OS as being in a hang state if it ceases to schedule tasks. This failure model is similar to the one introduced in [43]. In multiprocessor systems, it is possible for the OS to experience a hang on a proper subset of available CPUs. If that happens, we say that OS is in a *partial hang state*, as opposed to a full hang state, in which the OS is hung on all CPUs.

An example of a software bug that causes hangs in the OS kernel is a missing unlock (i.e., release) of a spinlock in an exit path of a kernel critical section. All threads that try to acquire this lock after the buggy exit path has been executed end up in a hung state. If the hung kernel thread is in a non-preemptible code section (e.g., either the kernel itself is non-preemptible, or the thread has purposely disabled preemption), the kernel hangs on the CPU that is executing the hung thread. The hung thread may also be holding other locks, which can cascade into hanging of more threads. In a multiprocessor system a partial hang usually results in a full hang. The kernel stays in a partial hang state until the hang propagates to all available CPUs. However, if the kernel has no other lock dependencies with the hung threads, it can stay in the partial hang state until it gets shut down or rebooted.

Distinguishing between OS partial and full hang is important for two reasons. (i) Previous OS hang detection approaches use external probes, e.g., heartbeats, to detect OS hangs. In a multiprocessor system, mechanisms to generate heartbeats may not be affected by a partial hang, and would continue to report error-free conditions. (ii) Detecting partial hangs results in a shorter detection latency, as all full hangs are preceded by a partial hang. The Guest OS hang detection (GOSHD) module discussed in this section detects both partial and full OS hangs.

### 4.1.2 GOSHD Mechanism

GOSHD uses the thread dispatching mechanism discussed in Section 3.3.1 to monitor the VM's OS scheduler. The `EPT_VIOLATION` and `CR_ACCESS` mechanisms in HAV guarantee that GOSHD can capture all context switch events. If a vCPU does not generate any switching events for a predefined threshold time, GOSHD declares that the guest OS is hung on that vCPU. Because the vCPUs are monitored independently of each other, GOSHD can detect both partial hangs and full hangs. From GOSHD's perspective, guest tasks are scheduled independently on each vCPU. Since GOSHD monitors the absence of context switching events to detect hangs, it is important to properly determine the threshold after which it is safe to conclude that the OS is hung on a vCPU. If this threshold is shorter than the time between two consecutive context switches, GOSHD generates false alarms. In order to be safe and fairly conservative, we profiled the guest OS to determine the maximum scheduling time slice, and set the threshold to be twice the profiled time. The numbers are usually on the order of milliseconds, or at most seconds, and are quicker compared to other hang detection techniques, such as heartbeat, or timer watchdogs, which frequently have detection times on the order of tens of seconds or minutes.

### 4.1.3 Functional Evaluation

#### Experimental Setup

The experiments were conducted on a guest VM with two vCPUs and 1024MiB of RAM. For the guest OS, we used the default build of SUSE Enterprise Linux Server 11 SP1, with and without kernel preemption enabled. The profiled maximum scheduling timeslice in both cases was two seconds, and hence the hang detection threshold was set to four seconds.

#### Experimental Methodology

In order to assess the hang detection capabilities of GOSHD, we used the fault injection framework proposed in [47]. As indicated in [47], one of the common causes of system hangs is improper implementation and invocation of locking mechanisms (e.g., spinlocks, reader/writer locks) that protect access to shared data structures in the kernel. Based on those findings, the authors of [47] identified four causes of hang failures: missing spinlock releases, wrong spinlock orderings, missing unlock/lock pairs, and missing interrupt state restorations. We further extended that concept to inject transient and persistent faults. A transient fault is only activated once when the fault location is first executed. Conversely, a persistent fault is activated every time the fault location is executed. Fault injection was repeated with different types of workloads running on the guest system:

- **Hanoi Tower**: "Tower of Hanoi" recursive program.
- **make -j1**: serial compilation of libxml.
- **make -j2**: compilation of libxml with two tasks in parallel.
- **HTTP server**: serving of an HTTP load from ApacheBench, which ran on a separate machine.

The first step of a fault injection experiment is to identify the injection location(s). We chose to inject faults into core functions of the Linux kernel and into frequently used kernel modules, such as ext3, char, and block. By profiling the kernel using the above workloads, we identified 374 locations on the execution path of the kernel to inject faults.

For each fault location, we started from a clean VM and then injected a fault while running the workload. There were five possible outcomes from each injection:

- **Not Manifested**: The fault was injected, but no observable failure was detected.
- **Not Detected**: A fault was injected, the VM was non-responsive but GOSHD did not report a vCPU hang.
- **Not Activated**: A fault was injected, but the workload did not execute the code that contained the fault.
- **Partial Hang**: At least one vCPU was still operational after 10 minutes (roughly twice the longest failure-free execution of the workloads) from the time a hang was detected on another vCPU.
- **Full Hang**: All vCPUs hung within 10 minutes after hang was detected on the first vCPU.

Detection Coverage Results

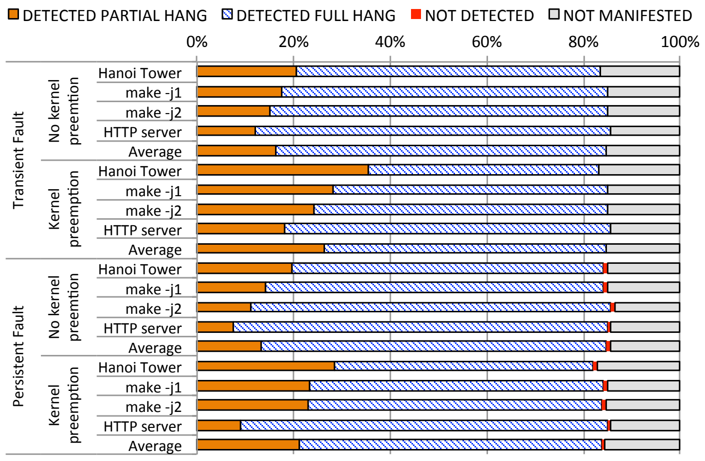


Figure 4.1: Guest OS hang detection coverage.

Figure 4.1 summarizes the detection coverage and percentages of partial and full hangs detected by GOSHD. About 82% of injected faults manifested as hangs. Overall, GOSHD missed 24 failures across all experiments, which resulted in 14,720 failures (17,952 injections $\times$ 0.82 manifested faults) or *a hang detection coverage of 99.8%.*

Further analysis of the misclassified failures indicates that the failures were caused by a fault location that was repeatedly activated by the guest SSH server, which was used by our external probe to check for false alarms by GOSHD. As a result, although the SSH probe reported hangs, the kernel and other processes on the VM still executed normally.

On average, 18% to 26% of faults caused partial hangs on the non-preemptible and preemptible kernels, respectively. Those significant numbers emphasize the importance of partial hang detection. In many partial hang cases, the VM was still accessible from outside (e.g., via SSH connections). That demonstrates the ineffectiveness of hang detection methods such as heartbeats, as the process/thread responsible for generating a heartbeat can still be fully operational and will continue to report that the system is as well.

Transient faults caused slightly more partial hangs than permanent faults did in single-task workloads (Hanoi Tower and make -j1), but significantly more partial hangs in concurrent multi-tasking workloads (make -j2 and HTTP server), because persistent faults can be reactivated and cause more independent hanging threads.

Kernel preemption does not appear to help prevent a hang due to spinlocks, as most critical sections in the kernel are non-preemptible. However, preemption does reduce the number of full hangs. For example, consider two tasks $T_1$ and $T_2$ sharing a user-level lock $l_u$. While holding $l_u$, task $T_1$ hangs because of our injection into a kernel spinlock. Task $T_1$ cannot be preempted because it is executing in a non-preemptible critical section (causing a partial hang). Now let us assume that task $T_2$ attempts to acquire $l_u$. In the non-preemptible kernel, task $T_2$ will hang as well, thus causing a full hang. But in the preemptible kernel, task $T_2$ can be preempted, and therefore the kernel remains in a partial hang.

### Detection Latency Results

Detection latency measures how quickly a detector can identify a problem. GOSHD raises an alarm when it finds that the guest OS scheduler has not scheduled processes for a predefined time. Therefore, GOSHD's minimal detection latency is that threshold (four seconds in our experiments). Specifi-

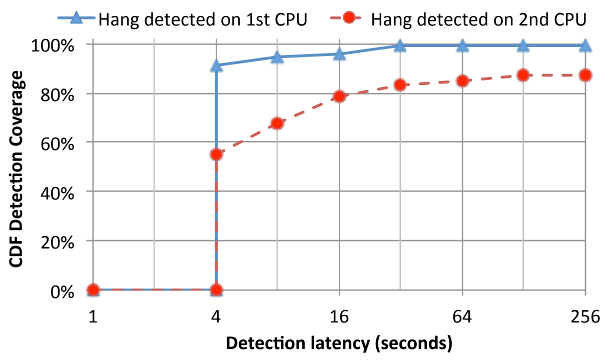


Figure 4.2: Guest OS hang detection latency. The blue line (with triangle markers) reflects the latency of detecting the first hang of a two vCPU VM. The red dashed line (with circle markers) reflects the latency of partial hangs.

cally, detection latency represents the time between fault activation and the moment GOSHD raises an alarm. Note that the guest OS is not necessarily hung at the moment the fault is injected. Figure 4.2 shows the detection latency of GOSHD for the same set of experiments described previously. Figure 4.2 demonstrates how partial hang detection helps reduce full hang detection latency. The blue line (triangles) shows that GOSHD can detect more than 90% of hangs after four seconds and all hangs within 32 seconds. Meanwhile, the red line (circles) shows that only 54% of hangs result in a full hang after four seconds. Many full hangs can be detected tens of seconds ahead through the use of partial hang detection.

## 4.2 HRKD – Hidden Rootkit Detection

### 4.2.1 Threat Model

Rootkits are malicious computer programs created to hide other programs from system administrators and security monitoring tools. For example, users cannot see a hidden process or thread via common administrative tools, such as Task Manager, PS, or TOP. Autonomic security scanning tools can also be bypassed simply because their inspection lists do not contain the hidden programs.

There are many existing techniques to hide a process, such as Direct Kernel Object Manipulation (DKOM) [48], physical memory manipulation [49], and dynamic kernel code manipulation [45]. For example, using those techniques, a rootkit can stealthily detach the data objects belonging to the malicious programs from their usual lists (e.g., remove a `task_struct` object from Linux's `task_list`). Therefore, a normal list traversal cannot reveal the detached object. As exemplified by previous studies [16, 45, 17], well-crafted rootkits can escape the detection of guest OS invariant-based scanning tools.

### 4.2.2 Detection Technique

Our HRKD module employs the context switch monitoring (Section 3.3.1) methods to inspect every process/thread that uses the vCPU, regardless of how kernel objects are manipulated. Each time a process or a thread is scheduled to use CPUs, it is intercepted by the module for further inspection. This interception defeats hidden malware; it puts malicious programs back on the inspection list.

Furthermore, HRKD uses architectural invariants to reliably infer the intercepted thread's state. Without architectural invariants, the inspection must start from an OS state, e.g., the head of the task_list, which may already be circumvented by a rootkit. With architectural invariants, the inspection starts from an architectural state, e.g., the TR or RSP registers, which directly point to the data structures used by hardware to execute each thread or process.

In order to detect a hidden user process or thread, the *process counting algorithm* (Fig. 3.1A) or *thread switch interception algorithm* (Fig. 3.1B) can be used. These algorithms are independent of the method by which the guest OS manages process-related data structures, because they rely only on architectural invariants. Inspection starts from the `CR3` or `TR` registers. Therefore, the observed number of processes always reflects the exact number of running processes. This is a trusted view that can be cross-validated against other views, e.g., a view from existing VMI tools or views from in-guest utilities, which may be the target of rootkits. Discrepancies between these views reveal the presence of hidden user processes and threads.

### 4.2.3 How Can a Rootkit Hide from HRKD?

A rootkit can hide from our HRKD by suppressing `CR3` access (for user-level rootkits) or `RSP0` access (for kernel-level rootkits) VM Exits. It can do so by *reusing the `CR3` (virtual address space) or `RSP0` (kernel stack)* of an existing process or kernel thread. Such attacks are called code injection attacks, which are not actually rootkits. Nevertheless, our HRKD is not designed to detect this class of attack.

### 4.2.4 Functional Evaluation

We tested HRKD on a variety of OSes and HRKD detected the presence of malware against all tested real-world rootkits.[1] On Windows, the tested rootkits included FU, HideProc, AFX, HideToolz, HE4Hook, and BH. HRKD's process counting technique showed additional processes beyond those reported by the Task Manager. On Linux, HRKD was able to discover all tested kernel-level rootkits: Ivyl's, Enyelkm 1.2, SucKIT, and PhalanX. Table 4.1 summarizes the results.

Since HRKD's process counting technique relies only on architectural invariants, it worked properly for all tested OSes, namely Windows XP, Vista, 7, and Server 2008, and various distributions of Linux kernel 2.6, without any adjustment. In addition, the detection capability of that technique was not affected by the implementation or strategy used by rootkits. In fact, the rootkits we evaluated employed a variety of hiding techniques, ranging from DKOM to system call hijacking (see Table 4.1). Thus, HRKD will be able to detect future hidden rootkits, even if they use novel hiding mechanisms.

## 4.3 PED – Privilege Escalation Detection

### 4.3.1 The Three Ninjas

Ninja [19] is a real-world privilege escalation detection system that uses passive monitoring. Ninja is included in the mainline repository for major

[1] We modified some rootkits' source code so they could work properly on our tested OS versions.

Table 4.1: Real-world rootkits evaluated with HRKD (All were detected)

| **Rootkit** | **Target OS** | **Hiding Technique(s)** |
|---|---|---|
| FU | Win XP, Vista | DKOM |
| HideProc | Win XP, Vista | DKOM |
| AFX | Win XP, Vista | Hijack system calls |
| HideToolz | Win XP, Vista, 7 | Hijack system calls |
| HE4Hook | Win XP | Hijack system calls |
| BH-Rookit-NT | Win XP, Vista | Hijack system calls |
| Ivyl's Rootkit | Linux >2.6.29 | Hijack system calls |
| Enyelkm 1.2 | Linux 2.6 | kmem, Hijack system calls |
| SucKIT | Linux 2.6 | kmem, DKOM |
| PhalanX | Linux 2.6 | kmem, DKOM |

Linux distributions, including Debian variants like Ubuntu. Ninja periodically scans the process list to identify if a root process has a parent process that is not from an authorized user (i.e., not defined in Ninja's "magic" group). If so, the root process is flagged as privilege-escalated. Ninja optionally terminates such processes to prevent further damage to the system. In order to avoid mistakenly killing setuid/setgid processes, Ninja allows users to create a "white list" of legitimate executables that are not subjected to its checking rules. The interval between checks is configurable (1s by default).

We implement HT-Ninja, which utilizes HyperTap for detecting privilege escalation attacks. We reuse the OS-level Ninja's checking rules when looking for unauthorized processes and make the following changes:

*Transform passive monitoring to active monitoring.* We define the following events at which a process is checked: (i) *first context switch of each process*; and (ii) *every I/O-related system call* (e.g., open, read, write, and lseek). That ensures that we check before any unauthorized actions, e.g., file or network, are conducted.

*Using architectural invariants.* The original Ninja uses Linux's `/proc` filesystem to obtain information about running processes. HT-Ninja uses only hardware state, such as the `TR` and `CR3` registers, to identify current running processes. HT-Ninja derives OS-specific information, such as User ID (uid) and Effective User ID (euid), from the `TSS` structure and `RSP` register, which can be combined to obtain the exact `thread_info` and `task_struct` objects of each process.

### 4.3.2 Functional Evaluation

#### Illustrating Attacks on Ninja

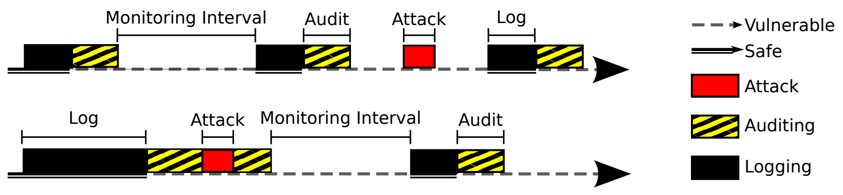


Figure 4.3: *Top*: Transient attack, the attacker attacks when a passive monitor is not logging. *Bottom*: Spamming attack, the attacker causes an attack to go undetected by creating extra work for both the logger and auditor.

Here, we intend to use Ninja only to demonstrate the limitations of passive monitoring, and are not criticizing its checking rules. We evaluated two passive-monitoring versions of Ninja: an original in-OS version (O-Ninja) and our modified version (H-Ninja), which was implemented at the hypervisor-level using *traditional VMI*. Later on, we will compare those two implementations against our active monitoring HT-Ninja. But first, we demonstrate four attacks that can bypass passive monitoring mechanisms:

*Transient attacks*: We used two real privilege escalation exploits, namely a glibc vulnerability (CVE-2010-3847) [50] and a kernel out-of-bounds error (CVE-2013-1763) [51] to obtain a terminal with root privileges. Ninja can easily detect the privilege escalated terminal if it remains in the system. However, when we terminated a process right after it finished an operation (e.g., copying sensitive data), both Ninja versions were unaware of the existence of the terminal, as its lifetime was short. See the top of Fig. 4.3 for an illustration.

*Side channel attacks*: We exploited the Linux `/proc` file system as a side channel [52] to determine Ninja's monitoring interval and when each check would be performed. Specifically, the file `/proc/PID/stat` allowed us to determine the current state of a process (e.g., Sleep or Running) and the current instruction pointer (EIP). By observing Ninja's process state, we could accurately measure the monitoring interval, i.e., how long Ninja was in the Sleep state and when the next check would be performed. With the measured interval and checking time, we could launch transient attacks that

Table 4.2: Predicting Ninja's monitoring interval (seconds)

| Ninja's Interval | Predicted Mean | Min | Max | SD |
|---|---|---|---|---|
| 1 | 1.00039 | 1.00006 | 1.00367 | 0.00071 |
| 2 | 2.00023 | 1.99998 | 2.00164 | 0.00038 |
| 4 | 4.00025 | 3.99986 | 4.00189 | 0.00040 |
| 8 | 8.00025 | 7.99951 | 8.00126 | 0.00035 |

avoided detection. However, that particular strategy did not work on H-Ninja, as it does not generate a `/proc` file in the target VM. Table 4.2 shows result of a trial of this method, in which each interval was sampled 30 times.

*Rootkit combined attacks*: In a more substantial attack, we combined a privilege escalation exploit with a rootkit, which was able to hide processes. After the terminal was escalated, we immediately ran the rootkit to prevent Ninja from discovering the presence of the terminal, bypassing both versions of Ninja.

*Spamming attacks*: We increased the execution time of the function that iterated over the process list by launching a large number of valid processes together with one privilege escalated process. The purpose was to increase the scanning time so that the escalated process can complete before the scan reached it. Note that a blocking H-Ninja is protected against this attack. See the bottom of Fig. 4.3 for an illustration.

### Active Monitoring with HT-Ninja

To show the benefits of HyperTap's active monitoring mechanism, we compared the detectability of the three versions of Ninja (O-Ninja, H-Ninja, and HT-Ninja) against real exploits, coupled with the attack strategies described in Section 4.3.2. It is worth mentioning that both O-Ninja and H-Ninja are vulnerable to DKOM rootkits, e.g., SucKIT, because they only use OS invariants.

Our experiments showed that O-Ninja with a 0-second checking interval was quickly defeated by a privilege escalation exploit CVE-2013-1763 [51] combined with spamming and a rootkit.[2] The attack was performed as follows: (i) a number of idle processes were created; (ii) the exploit code was

[2] `https://github.com/ivyl/rootkit`

run to grant root privileges to the current process; and (iii) with root privileges, the rootkit was installed to remove the escalated process from the process list. We repeated the attack 300 times and have timed the attack to take ∼4ms on an Intel(R) Core(TM)2 Duo CPU E8400 3GHz CPU. Without creating extra processes on a system with 31 processes running, O-Ninja can detect ∼10% attacks. When we introduced 100 idle processes, the detection probability was reduced to single-digit percentages (between 2 and 3%). With 200 or more idle processes, O-Ninja cannot detect any attacks (with the exception of outliers $\ll 1\%$).

To test H-Ninja, we used the same privilege escalation + rootkit combination as with O-Ninja. Since the attack was quick, a small value for the checking interval was needed. With an interval of 4 ms, H-Ninja could detect 100% of the attacks, and the detection dropped to near 60% at 8 ms. With an interval $>$ 20 ms, the detection probability became $<$ 5%.

Although an attacker with no access to side channels must rely on a certain amount of luck to defeat O-Ninja and H-Ninja, his/her probability of success can be increased by spamming (O-Ninja) or by reasoning that administrators would not wish to incur the penalty of scanning the process tree with a frequency in the single-digit milliseconds. Also, the speed of the attack at 4 ms was extremely naïve; a more sophisticated attacker should be able to do better.

Since it uses active monitoring, HT-Ninja was able to detect all attacks in all tested scenarios.

A main limitation of HT-Ninja, as well as O-Ninja and H-Ninja, is that they do not detect privilege escalation attacks that occur within the context of "white listed" processes. Those processes, many of which are setuid programs, are ignored by Ninja. An attack that compromises (e.g., using buffer overflow) and executes malicious code within the context of a white listed process would not be detected.

## 4.4 Other Uses of HyperTap

The logging capabilities presented in Section 3.3 can also be used to implement a wide variety of RnS monitors. For example, there is a class of security tools that depend on system call interception [53, 54, 55]. Failure detection based on machine learning, e.g. [40], can be applied to the events and states logged by HyperTap.

HyperTap could also be incorporated into the runtime memory safety technique proposed in [56]. That technique consists of two steps: (i) compiler analysis and instrumentation, to guide (ii) runtime memory safety checking. The latter step requires OS modification to intercept privileged operations, e.g., MMIO, MMU configuration, and context switching [57]. Since HyperTap supports those interceptions without altering the guest OS, it shows promise for being integrated with runtime checking. Such incorporation would allow a variety of RnS detectors to be implemented, such as detectors for silent data corruption, buffer overflow, and code injection. We leave that integration for future work.

## 4.5 Performance Evaluation

We conducted experiments to measure the performance overhead of individual HyperTap auditors as well as the combined overhead of running multiple auditors. We measured the runtime of the UnixBench[3] performance benchmark when (i) each auditor was enabled, and (ii) all three auditors are enabled. The target VM was a SUSE 11 Linux VM with 2 vCPUs and 1GiB of RAM. The host computer ran SUSE 11 Linux and the KVM hypervisor, with an 8 core Intel i5 3.07GHz processor and 8 GiB of RAM. The results were illustrated in Figure 4.4. The baseline is the execution time when running the workloads in the VM without HyperTap integrated, and the reported numbers are the average of five runs of the workloads.

In most cases, the performance overhead of running all three auditors simultaneously was (i) only slightly higher than that of running the slowest auditor, HT-Ninja, individually, and (ii) substantially lower than the summation of the individual overheads of all auditors. That result demonstrates the benefits of HyperTap's unified logging mechanism.

For the Disk I/O and CPU intensive workloads, all three auditors together produced less than 5% and 2% performance losses, respectively. The Disk I/O intensive workloads appear to have incurred more overhead than CPU intensive workloads because they generated more VM Exit events, at which

[3]`http://code.google.com/p/byte-unixbench/`

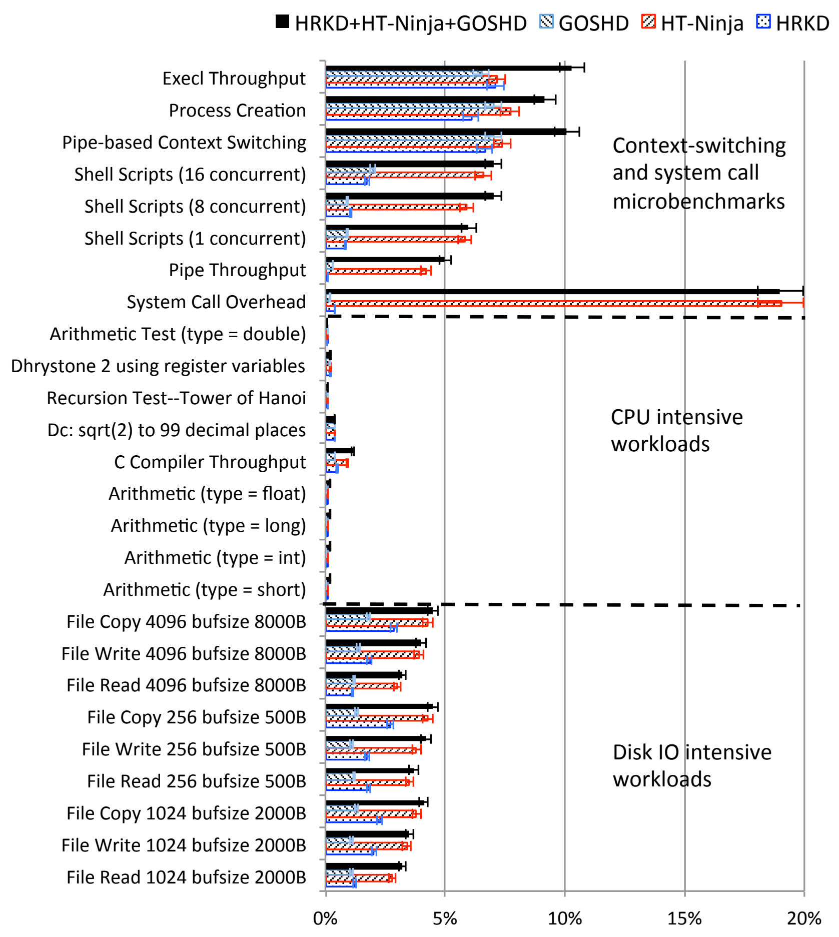


Figure 4.4: Measured performance overhead of HyperTap sample monitors. The workloads are run with three different configurations: 1) Both HRKD and HT-Ninja, 2) only HT-Ninja, and 3) only HRKD. Error bars indicate one standard deviation.

point some monitoring code was triggered.

For the context switching and system call micro-benchmarks, all three auditors together induced about 10% (or less) and 19% performance losses, respectively. It is important to note that those micro-benchmarks were designed to measure the performance of individual specific operations without any useful processing; they do not necessarily represent the performance overhead of general applications. The relatively high overhead was caused by the HyperTap routines enabled for logging those benchmarked operations. Since only HT-Ninja needs to log system calls, it was the primary source of the overhead in the system call micro-benchmark case.

## 4.6 Conclusion

The feasibility of the HyperTap framework was demonstrated through the implementation and evaluation of three monitors: Guest OS Hang Detection, Hidden RootKit Detection, and Privilege Escalation Detection. In all cases, the use of architectural invariants was central to the high quality and performance observed in the experiments. We presented additional analysis of the method so that other reliability and security monitors can be built on top of the HyperTap framework.

# CHAPTER 5

# HPROBES: DYNAMIC VIRTUAL MACHINE MONITORING USING HYPERVISOR PROBES

*This chapter introduces Hprobe, a framework that allows one to dynamically monitor applications and operating systems inside a VM. The Hprobe framework does not require any changes to the guest OS, which avoids the tight coupling of monitoring with its target. Furthermore, the monitors can be customized and enabled/disabled while the VM is running.*

## 5.1 Introduction

The HyperTap framework introduced in the previous chapters provides an efficient and hard-to-bypass event-driven monitoring mechanism. The key design of HyperTap is the reliance on a fixed set of hardware architectural invariants to capture guest OS's activities. While we have shown that this monitoring capability is effective to support an important set of reliability and security monitoring policies (see Chapter 4 for examples of the evaluated policies), there are still many cases in which monitors requires a more flexible means to place monitoring points, or hooks, to capture specific guest OS and applications' operational activities.

One class of active monitoring systems is a hook-based system, where the monitor places hooks inside the target application or OS [13]. A hook is a mechanism used to generate an event when the target executes a particular instruction. When the target's execution reaches the hook, control is transferred to the monitoring system where it can record the event and/or inspect the system's state. Once the monitor has finished processing the event, it returns control to the target system and execution continues until the next event. Hook-based techniques are robust against failures and attacks inside the target when the monitoring system is properly isolated from the target system.

We find *dynamic hook-based* systems attractive for dependability monitoring as they can be easily adapted: once the hook delivery mechanism is functional, implementing a new monitor involves adding a hook location and deciding how to process the event. In this case, dynamic refers to the ability to add and remove hooks without disrupting the control flow of the target. This is particularly important in real-world use, where monitoring needs to be configured for multiple applications and operational environments. In addition to supporting a variety of environments, monitoring must also be responsive to changes in those environments.

In this chapter, we present the Hprobe framework, a dynamic hook-based VM reliability and security monitoring solution. The key contributions of the Hprobe framework are that it: is loosely coupled from the target VM, can inspect both the OS and user applications, and it supports runtime insertion/removal of hooks. All of these aspects result in a VM monitoring solution that is suitable for running on an actual production system. We have built a prototype implementation using hardware-assisted virtualization that is integrated with the KVM hypervisor [6]. From our experiments, the overhead for an individual probe (the time between hook invocation and when control is returned to the VM) is 2.6 $\mu s$ on a modern server-class CPU. To demonstrate monitoring using the Hprobe framework, we have constructed an emergency security vulnerability detector, a heartbeat detector, and an infinite loop detector. While our prototype framework shares some similarities and builds on previous monitoring systems, these detectors could not have been implemented on any existing platform. All of these detectors were tested using real applications and exhibit low overhead ($\leq 5\%$).

## 5.2 Design

### 5.2.1 Hook-Based Monitoring

An illustration of a hook-based monitoring system adapted from the formal model presented in Lares [13] is shown in Figure 5.1. Hook-based monitoring involves a monitor takes control of the target after the target reaches a hook. In the case of hypervisor-based VM monitoring, the target is a virtual machine and the monitor can run in either the hypervisor [10], in a sepa-

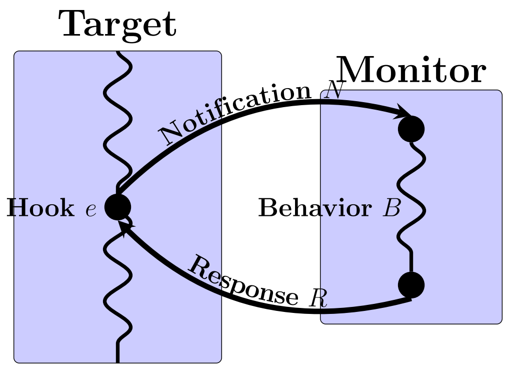


Figure 5.1: Hook-based monitoring. A hook triggers based on event $e$ and control is transferred to the monitor through notification $N$. The monitor processes $e$ with a behavior $B$ and returns control to the target with a response $R$.

rate security VM [13], or in the same VM [30]. Regardless of the separation mechanism used, one must ensure that the monitor is resilient to tampering from within the target VM and the monitor has access to all relevant state of that VM (e.g., hardware, memory, etc.). Furthermore, a VM monitoring system should be able to trigger on the execution of any instruction, be it in the guest OS or in an application.

If a monitoring system can capture all relevant events, it also follows that the monitoring system should be *dynamic*. This is important in the fast-changing landscape of IT security and reliability. As new vulnerabilities and bugs are discovered, one will inevitably need to account for them. The value of a static monitoring system decreases drastically over time unless periodic software updates are issued. However, in many VM monitoring solutions [8, 14, 13, 30], such software updates would require a hypervisor reboot or at the very least a guest OS reboot. These reboots result in system downtime whenever the monitor needs to be adapted. In many production systems, this additional downtime is unacceptable, particularly when the schedule is unpredictable (e.g., security vulnerabilities). Dynamic monitors can also provide performance improvement over statically configured monitoring: one can monitor only event of interest vs. a general class of events (e.g., a single system call vs. all system calls). Furthermore, it is possible to construct dynamic detectors that change during execution (e.g., a hook can be used to add or remove other hooks). Static monitoring systems also present a subtle design flaw: a configuration change in the monitoring system can affect the control flow of the target system (e.g., by requiring a restart).

In line with dynamism and loose coupling with the target system, the detector must also be simple in its implementation. If a system is overly complex and difficult to extend, the value of that system is drastically reduced as much effort needs to be expended to use that system. In fact, such a system will simply not be used. DNSSEC[1] and SELinux[2] can serve as instructive examples: while they provide valuable security features (e.g., authentication and access control), both of these systems were released around the year 2000 and to this day are still disabled in many environments. Furthermore, a simpler implementation should yield a smaller attack surface [58].

### 5.2.2 Design Principles

In light of the observation made in the previous section, we set the following design principles for a dynamic VM active monitoring system:

1. **Protection:** Monitoring should be impervious to attacks (e.g., hook circumvention) inside the VM. The authors of Lares [13] outline a formal model with potential attacks and security requirements for a hook-based monitoring system. Those requirements using the notation in Fig. 5.1 are: the notification $N$ should only be triggered on legitimate events, the state of the target should not change during monitoring, an attacker cannot modify the behavior $B$ of the monitor, and the response $R$ cannot be avoided by the target.

2. **Simplicity:** The monitoring system should be simple to implement and extend. In order to ease adoption and support cloud environments, it should not require any modification of the guest OS.

3. **Dynamism:** The monitoring system should be loosely coupled with the target. The target itself should be protected from changes in the monitoring system: reconfiguration can be expected to affect execution time, but it should not disrupt the control flow of the target (e.g., require a reboot or application restart). Furthermore, it should be possible to insert the hooks into both the target OS and its applications.

[1]https://tools.ietf.org/html/rfc2535
[2]https://www.nsa.gov/public_info/press_room/2001/se-linux.shtml

4. **Performance:** The monitoring system should have acceptable overhead for use in a production system.

We use these requirements as a guide to design a hook-based hypervisor monitoring framework that we call hypervisor probes or *Hprobes*. The hypervisor provides a convenient interface for isolating monitoring from the VM while maintaining full access to the target VM. The proposed framework allows one to insert and remove hooks into arbitrary locations inside the guest's memory (i.e., both the guest OS and user applications) at runtime. To demonstrate the effectiveness of our framework, we build a prototype and three monitors. Two of the monitors implement reliability techniques, and the third illustrates the simplicity of using Hprobes to rapidly produce a monitor that protects against a security vulnerability.

## 5.3 Prototype Implementation

### 5.3.1 Review Debugging with Software Interrupt `int3`

The x86 architecture offers multiple methods for inserting breakpoints, which are used in our prototype framework. We focus on the `int3` instruction as it is flexible and is not limited in the number of breakpoints that can be set. The `int3` instruction is a single byte opcode (`0xcc`) that raises a breakpoint exception (`#BP`). A debugger uses OS provided functionality (e.g., a system call like `ptrace()` [59] in Linux) to control and inspect the process being debugged. In order to insert a breakpoint, a debugger overwrites the instruction at the desired location with `int3`, and then saves the original instruction. When the breakpoint is hit and the `#BP` exception is generated, the OS catches the exception and notifies the debugger. At this point, the debugger has control of the process and can inspect the process's memory or control its execution, e.g., by single-stepping over subsequent instructions. More details can be found in Chapter 17 of the Intel Software Developer's Manual [4].

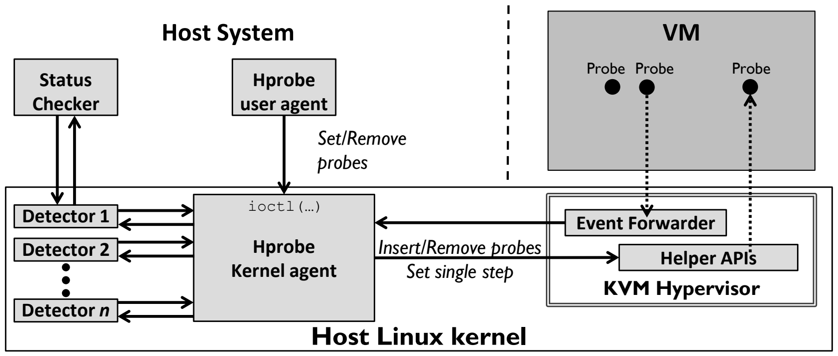


Figure 5.2: Hprobes integrated with the KVM hypervisor. The Event Forwarder has been added to KVM and communicates with a separate kernel agent through Helper APIs. Detectors can either be implemented as kernel modules in the Host OS or in user space by communicating with the kernel agent through `ioctl` functions.

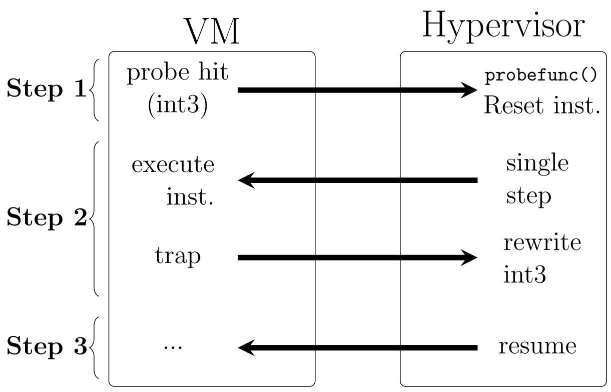


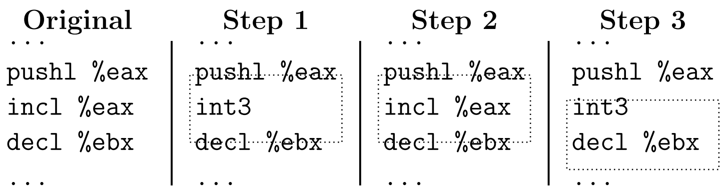
Figure 5.3: A probe hit in the Hprobe prototype. Right-facing arrows are VM Exits and left-facing arrows are VM Entries. When `int3` is executed, the hypervisor takes control. The hypervisor optionally executes a probe handler (`probefunc()`) and places the CPU into single-step mode. It then executes the original instruction and does a VM Entry to resume the VM. After the guest executes the original instruction, it traps back into the hypervisor and the hypervisor will write the `int3` before allowing the VM to continue as usual.

### 5.3.2 Integration with KVM

The Hprobe prototype was inspired by the Linux kernel profiling feature kprobes [60], which has been used for real-time system analysis [61]. The operating principle behind our prototype is to use VM Exits to trap the VM's execution and transfer control to monitoring functionality in the hypervisor. This implementation leverages hardware-assisted virtualization (HAV), and the prototype framework is built on the KVM hypervisor [6]. The prototype's architecture is shown in Figure 5.2. The modifications to KVM itself make up the Event Forwarder, which is a set of callbacks inserted into KVM's VM Exit handlers. The Event Forwarder communicates with a separate hprobe kernel agent using Helper APIs. The hprobe kernel agent is a loadable kernel module that is the workhorse of the framework. The kernel agent provides an interface to detectors for inserting and removing probes. This interface is accessible by kernel modules through a kernel API in the host OS (which is also the hypervisor since KVM itself is a kernel module) or by user programs via an `ioctl` interface.

The execution of an Hprobe based detector is illustrated in Figure 5.3 and Figure 5.4. A probe is added by rewriting the instruction in memory at the target address with `int3`, saving the original instruction, and adding the target address to a doubly-linked list of active probes. This process happens at runtime and requires no application or guest OS restart. As explained in Section 5.3.1, the `int3` instruction generates an exception when executed. With HAV properly configured, this exception generates a VM Exit event, at which point the hypervisor intervenes (Step 1). The hypervisor uses the Event Forwarder to pass the exception to the Hprobe kernel agent, which traverses the list of active probes and verifies that the `int3` was generated by an Hprobe. If so, the Hprobe kernel agent reports the event and optionally calls an *hprobe handler* function that can be associated with the probe. If the exception does not belong to an Hprobe (e.g., it was generated by running gdb or kprobes inside the VM), the `int3` is passed back to KVM to be handled as usual. Each Hprobe handler performs a user-defined monitoring function and runs in the Host OS. When the handler returns (a deferred work mechanism can also be used to support non-blocking probes, if desired), the hypervisor replaces the `int3` instruction with the original opcode and put the CPU in single-step mode. Once the original instruction executes, a single-step

(#DB) exception is generated, causing another VM Exit event [4] (Step 2). At this point, the Hprobe kernel agent rewrites the `int3`, performs a VM Entry, and the VM resumes its execution (Step 3). This single-step and instruction rewrite process ensures that the probe is always caught. If one wishes to protect the probes from being overwritten by the guest, the page containing the probe can be write-protected. Although this prototype was implemented using KVM, the concept will extend to any hypervisor that can trap on similar exceptions. Note that instead of `int3`, we could use any other instruction that generates VM Exits (e.g., hypercall, illegal instruction, etc.). We chose `int3` since it is well supported and has a single-byte opcode.



| **Original** | **Step 1** | **Step 2** | **Step 3** |
|---|---|---|---|
| `...` | `...` | `...` | `...` |
| `pushl %eax` | `pushl %eax` | `pushl %eax` | `pushl %eax` |
| `incl %eax` | `int3` | `incl %eax` | `int3` |
| `decl %ebx` | `decl %ebx` | `decl %ebx` | `decl %ebx` |
| `...` | `...` | `...` | `...` |

Figure 5.4: Assembly pseudocode demonstrating what an Hprobe looks like in the VM's memory before adding a probe (left frame) and during a probe hit (right three frames). The dashed box indicates the VM's current instruction.

### 5.3.3 Building Detectors

As mentioned in the previous section, Hprobes can be controlled via an `ioctl` interface or a kernel API. Both interfaces distinguish between probes that are inserted into guest kernel space and guest user space. That is because while the OS always maps the kernel space pages at the same address for all virtual address spaces, each user program has its own set of pages. User space probes require the Page Directory Base Address (from the CR3 register on x86) to translate a guest virtual address into a guest physical address. Once we know the guest physical address, we can overwrite the instruction at that address and insert probes into the address space of a particular process. However, the mapping of an OS-level construct like a running process to hardware paging structures is not readily available from the hypervisor due to the semantic gap between the VM and the hypervisor. Therefore, we use libVMI to obtain the value of the CR3 register corresponding to the target process's virtual address space [62]. This allows us to translate the virtual address of a probe location (which can be obtained from dynamic/static analysis, or by inspecting the application's symbol table) to a guest physical address that can be used to add a probe.

If one wishes to insert a probe into a user application, however, there exists another challenge. Unlike the guest OS, the pages of a running application's code may not be resident in memory at all times. That is, during an application's lifetime, some of its code may reside on disk. When execution reaches a page that is not resident, the OS will bring that page into memory. This means that the hypervisor may not be able to insert probes directly into all locations of the program at all times (i.e., it would have to wait for the OS to bring certain pages into memory). This situation arises particularly during application startup. In this case, the OS uses a demand paging mechanism in which the pages belonging to the application reside on disk until the application attempts to access one of those pages. Therefore, if the page containing the target location for a probe has not yet been accessed, a translation for guest physical address to guest virtual address will not exist. In order to support probes for user programs, this situation must be resolved so that the Hprobe framework can guarantee that once a probe has been added through the APIs, it will get called on the next invocation of the instruction at the probe's desired location.

One approach to solving the problem of having target code paged out is to wait until the OS naturally brings the necessary page into memory. As mentioned in Section 2.2, recent versions of x86 hardware assisted virtualization (HAV) use two-dimensional page tables, and do not require VM Exits for all page table updates. Therefore, in order to trap a page table update when using EPT, one must remove access permissions from EPT entries to induce an `EPT_VIOLATION` VM Exit event. In this case, we remove write permissions from the *guest physical page* corresponding to the ***guest page table entry*** that refers to the *guest virtual page* for the intended probe location. We remind the reader that in this case the page itself is not yet present in the guest OS, and therefore a translation from *guest virtual address* to *guest physical address* does not exist in the guest OS paging structures. When an `EPT_violation` corresponding to our protected ***guest page table entry*** occurs (indicating that the page containing the probe location is now in memory), we put the CPU into single-step mode. After the instruction writing to the guest page table executes, we can insert the probe by performing the usual translations and traversing the guest paging structures. This process of using page protection to insert probes into non-resident locations is described in Figure 5.5. Note that we could improve performance slightly by avoiding the single-step and decoding the trapped instruction that caused the `EPT_VIOLATION`. In practice, however, this paged-out situation only occurs once during the lifetime of the program (unless a page is swapped out, in which case disk latency would dominate VM Exit latency) and the performance gain would be negligible.

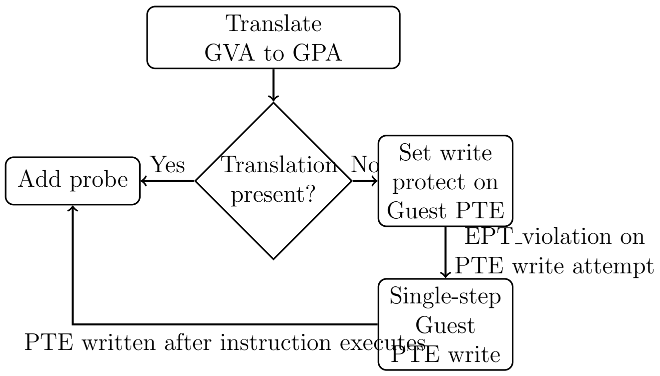

Translate GVA to GPA
Translation present?
Yes
No
Add probe
Set write protect on Guest PTE
EPT_violation on PTE write attempt
Single-step Guest PTE write
PTE written after instruction executes

Figure 5.5: How a user space probe is added. A guest virtual address (GVA) for the probe's location must be translated into a guest physical address (GPA). If the translation fails because the page is not present, we write protect the EPT page containing the guest page table entry (PTE) for that GVA. When the guest OS attempts to update the guest page table, the Hprobe kernel agent is notified via an `EPT_violation` and sets single step mode. After the single-step, the translation succeeds, and the probe is added.

Often times when monitoring, it is necessary to not only be aware of events in the VM (e.g., an instruction at a particular address was executed), but also the state of the VM (e.g., registers, flags, etc...). When inserting an hprobe from within the hypervisor (i.e., using a kernel module in the Host OS), the hprobe kernel agent passes a pointer to a structure containing vCPU state to the hprobe handler. These privileged probe handlers can use this structure to decode additional information or possibly modify the state of the VM to mitigate a failure or vulnerability.

### 5.3.4 Discussion

Our use of `int3` to generate an exception utilizes hardware enforcement of event generation: there is no dependence on any functionality inside the guest OS. This allows the hprobe hooking mechanism to be used on any guest OS supported by the hypervisor. Since the majority of the work is done outside of the hypervisor modifications (i.e., all of the heavy lifting is done inside of the kernel agent), the system can be ported to other hypervisors that support trapping on `int3`.

When reflecting on the requirements set forth in Section 5.2, we observe that the hprobe framework satisfies those requirements:

1. **Protection**: By using an out-of-VM approach that is enforced by HAV, our hooks cannot be circumvented. Furthermore, we can use memory protection in the hypervisor to prevent probes from being modified (or hide them by read protecting them).

2. **Simplicity**: Modifications to introduce the Event Forwarder and Helper APIs to KVM add only 117 source-lines-of-code (SLOC) and the kernel agent is 703 SLOC. The simple API allows monitors to be developed quickly and most detectors can be based on a common template (e.g., build one detector by reusing a majority of the code from a previous one). As an anecdotal example, most of the example detectors presented in Section 5.4 required only two hours of programming to be fully functional. Hprobes can be used on an unmodified guest OS.

3. **Dynamism**: Our API allows for the insertion and removal of probes at runtime without disrupting the control flow of the target VM. Furthermore, unique to hook-based VM monitoring systems, we support application level monitoring through user space probes.

4. **Performance**: While we require multiple VM Exits, we find that for our test applications and use cases, the performance is acceptable and worth the value added in the previous two dimensions. See Section 5.5 for analysis and details.

This prototype satisfies the protection requirements adapted from Lares [13] in Section 5.2.2. The notification $N$ is only delivered if events occur legitimately (spurious `int3`s are ignored by the kernel agent). The context information of the event (the VM's state at event $e$) cannot be modified during hprobe processing since the hypervisor is in control. The security application (e.g., a `probefunc()`) runs inside the hypervisor and therefore, its behavior $B$ cannot be altered by the VM. Additionally, the effects of any response $R$ from the hypervisor are enforced since the hypervisor has full control over the target VM. Since hprobes configure VM Exits to occur on `int3`, one could imagine a Denial-of-Service (DOS) attack based on causing VM Exits using spurious `int3` instructions. We note that hprobes do not present a new DOS threat and that if an attacker were interested in such an attack, he or she can perform it using existing functionality (e.g., using the `vmcall` instruction).

While using the hprobe framework does require modifications to the hypervisor, these modifications are small and robust across multiple versions of KVM and the Linux kernel. During the course of this project, we used the diff-match-patch libraries[3] to migrate the Event Forwarder and Helper APIs between KVM versions. We have tested hprobes on OpenSUSE 11.2, CENTOS7, Gentoo with kernel version 3.18.7, Ubuntu 12.04, and Ubuntu 14.04. The hprobe kernel agent is written to be version agnostic (e.g., with `#ifdef` macros for kernel version specific constructs like `unlocked_ioctl`).

### 5.3.5 Limitations

This prototype is useful for a large class of monitoring use cases, however it does have a few limitations. Namely,

1. Hprobes only trigger on instruction execution. If one is interested in monitoring data access events (e.g., trigger every time a particular address is read from/written to), hprobes do not provide a clean way to do so. One would need to place a probe at every instruction that modifies the data (potentially every instruction that modifies any data if addresses are affected by user input). More cleanly, one could use an hprobe at the beginning and end of a critical section to turn on and off page protection for data relevant to that critical section, capturing the events in a manner similar to livewire [8], but with the flexibility of hprobes. We are considering this in future work.

[3]https://pypi.python.org/pypi/diff-match-patch/

2. Hprobes leverage VM Exits, resulting in non-optimal performance. This tradeoff is worth the simpler, more robust implementation with its trust rooted in HAV.

3. Probes cannot be fully hidden from the VM. Even with clever EPT tricks to hide the existence of a probe when reading from its location, a timing side channel would still exist since an attacker could observe that the probed instruction takes longer than expected to complete.

## 5.4 Detectors

In this section, we present sample reliability and security detectors built upon the hprobe prototype framework. These detectors are unique to the hprobe framework and cannot be implemented on any other current VM monitoring system.

### 5.4.1 Emergency Exploit Detector

Most systems operators fear zero-day vulnerabilities as there is little that can be done about them until the vendor/maintainer of the software releases a fix. Furthermore, even after a vulnerability is made public, a patch takes time to be developed and must be put through a QA cycle. This can further be exacerbated in environments with high availability concerns and stringent change control requirements: even if a patch is available, many times it is not possible to restart the system or service until a regular maintenance window. This leaves operators with a difficult decision: risk damage from restarting a system with a new patch or risk damage from running an unpatched system.

Consider the CVE-2008-0600 vulnerability that was resulted in a local root exploit through the `vmsplice()` system call [63, 64]. This example represents a highly dangerous buffer overflow since a successful exploit allows one to arbitrarily execute code in ring 0 using a program that is publicly available on the Internet. Since this exploit involves the base kernel code (i.e., not a loadable module), patching it would require installing a new kernel followed by a system reboot (or without a reboot using techniques such as [65, 66]). As discussed earlier, in many operational cases a system reboot or OS patch can only be conducted during a predetermined maintenance window. Furthermore, many organizations would be hesitant to run a fresh kernel image on production systems without having gone through a proper testing cycle first.

The `vmsplice()` system call is used to perform a zero-copy map of user memory into a pipe. At a high level, the CVE-2008-0600 `vmsplice()` constructs specially crafted compound page structures in userspace. A compound page is a structure that allows one to treat a set of pages as a single data structure. Every compound page structure has a pointer to a destructor function that handles the cleanup of those underlying pages. The exploit works by using an integer overflow to corrupt the kernel stack such that it references the compound page structures crafted in userspace. Before calling `vmsplice()`, the exploit closes the pipe, so that when the system call runs it deallocates the pages, resulting in calling the compound pages' destructor function. The destructor is set to privilege escalation shellcode that allows an attacker to hijack the system.

The CVE-2008-0600 exploit hinges on an integer overflow in one of the system call arguments - a pointer to a `struct iovec` that contains the member `iov_len`, which is set to `ULONG_MAX` by the exploit. Since Linux uses registers to hold the system call number as well as arguments for system calls [67], we could use classical system call monitoring/tracing tools to detect this exploit [68, 55]. We can watch whenever a system call is invoked and check for the correct system call number and parse arguments to detect an integer overflow attempt. However, since hprobes are dynamic, we can set a probe to trigger only on the `sys_vmsplice()` function (that is called after the system call assembly linkage). This ensures that only the execution path of the `vmsplice()` system call is inspected as opposed to all system calls (as in traditional system call tracing). At this point in the system call invocation, the function just uses the regular compiler function calling convention (in most instances of the Linux kernel, the gcc convention) and the arguments are on the stack. Either way, we can use hprobes to obtain these arguments. Essentially, one needs to ensure that `iov_len` will not cause overflow. Depending on the environment, the operator can choose how to handle the detected exploit. One could send an alert, simply modify `iov_len` to a benign value that causes `vmsplice()` to fail, or take a more drastic action (such as killing the process or VM) if desired.

The emergency detector works by checking the arguments of a system call for a potential integer overflow. This differs in functionality from the upstream patch,[4] which checks if the memory region (specified by the `struct iovec` argument) is accessible to the user program. One could write a probe handler that performs a similar function by checking if all of the region referred to by the `struct iovec` pointer + `iov_len` is in the appropriate range (e.g., by walking the page tables belonging to that process). However, a temporary measure to protect against an attack should be as lightweight and simple as possible to avoid unpredictable side effects. One major benefit of using an hprobe handler is that developing this detector does not require a deep understanding of the vulnerability: the developer of the emergency detector only needs to understand that there is an integer overflow in an argument. This is far simpler than developing and maintaining a patch for a core kernel function (a system call), especially when reasoning about the risk of running a home-patched kernel (a process that would void most enterprise support agreements).

Our solution uses a monitoring system that resides outside of the VM and relies on a hardware-enforced `int3` event. A would-be attacker cannot circumvent this event without having first compromised the hypervisor or having modified the guest's kernel code. This could be done with a code injection attack that causes a different `sys_vmsplice()` system call handler to be invoked. However, it is unlikely that an attacker who already has the privileges necessary for code injection into the kernel would have anything to gain by exploiting a local privilege escalation vulnerability. While this detector cannot defeat an attacker that has previously obtained root access, its ease of rapid deployment sufficiently mitigates this risk. Since no reboot is required and the detector can be used in a "read-only" monitoring mode (only reporting the attack vs. taking an action), the risk of using this detector on a running production system is minimal. To test the CVE-2008-0600 detector, we used a CENTOS5 VM (the exploit was discovered while the source-equivalent Red Hat Enterprise Linux 5.0 OS was in production) and the publicly available exploit. As an unprivileged user, we ran an exploit script on the unpatched OS and were able to obtain root access. With the monitor in place, all attempts to obtain root access using the exploit code

[4]https://gitorious.org/kernel-linux/linux-stable/commit/af395d8632d0524be27d8774a1607e68bdb4dd7f

were detected.

### 5.4.2 Application Heartbeat Detector

One of the most basic reliability techniques used to monitor computing system liveness is a heartbeat detector. In that class of detector, a periodic signal is sent to an external monitor to indicate that the system is functioning properly. A heartbeat serves as an illustrative example for how an hprobe-based reliability detector can be implemented. Using hprobes, we can construct a monitor that directly measures the application's execution. That is, since probes are triggered by application execution itself, they can be viewed as a mechanism for direct validation that the application is functioning correctly. Many applications execute a repetitive code block that is periodically reentered (e.g., a Monte Carlo simulation that runs with a main loop, or an http server that constantly listens for new connections). If one profiles the application, it is possible to determine a period (in units of time or using a counter like the number of instructions) at which this code block is reentered. During correct operation of the application, one can expect that the code block will be executed at the profiled interval.

The hprobe-based application heartbeat detector is built on the principle described in the previous paragraph and illustrated in Fig 5.6. This test detector is a kernel module that is installed in the Host OS (i.e., one of the detectors on the left side of Fig. 5.2). An hprobe is inserted at the start of the code block that is expected to be periodically reentered. When the hprobe is inserted, a delayed workqueue[5] is scheduled for the timeout corresponding to the reentry period for the code block. When the timeout expires, the workqueue function is executed and declares failure (if the user desires a more aggressive watchdog style detector, it is possible to have the hprobe handler perform an action such as restart the application or VM). During correct operation (i.e., when the hprobe is hit), the workqueue is canceled and a new workqueue is scheduled for the same interval, starting a new timeout period. This continues until the application finishes or the user no longer desires to monitor it and removes the hprobe. If having an hprobe hit on every iteration of the main loop is too costly, one can ensure that the

[5]http://www.makelinux.net/ldd3/chp-7-sect-6

probe active for an acceptable time interval and it can be added/removed until desirable performance is achieved (the detection latency would still be low as a tight loop would have a small timeout value).

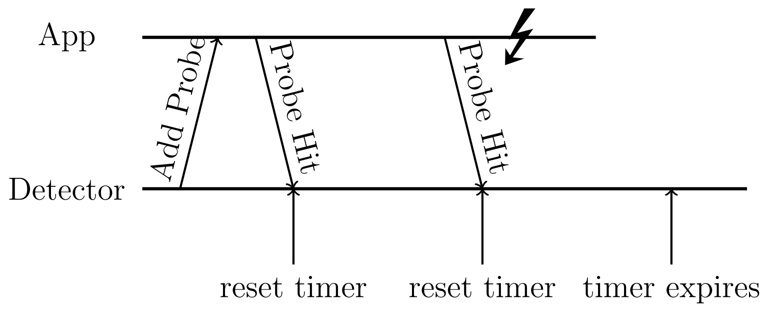


Figure 5.6: Application heartbeat detector. A probe is inserted in a critical periodic section of the application (e.g., the main loop). During normal execution, a timer is continuously reset. In the presence of a failure (such as an I/O hang), the timer expires and failure is declared.

We use the open-source Path Integral Quantum Monte Carlo (pi-qmc) simulator [69] as a test application.[6] This application represents a long-running scientific program that can take many hours or days to complete. As is typical with scientific computing applications, pi-qmc has a large main loop. Since Monte Carlo simulation involves repeated sampling and therefore repeated execution of the same functions, we only need to run the main loop a handful of times to determine the time per iteration. After determining the expected duration of each iteration, we set the heartbeat to timeout to the twice the expected value, set the detector to a statement at the end of the main loop, and injected hangs (e.g., SIGSTOP) and crashed the application (e.g., SIGKILL). All crashes (including VM crashes since the timer executes in the hypervisor) were detected.

### 5.4.3 Infinite Loop Detector

Infinite loops are a common failure that can cause process hangs. When considering proper execution of a loop in a program (that is not the main loop), the number of instructions executed in a given block of code usually falls into a fixed range, with the upper bound being the worst case execution time (WCET) [20]. Determining the WCET is a well studied problem in

[6]Available at: http://phys-tools.github.com/pi-qmc/

real-time systems, and solving it is beyond the scope of our work. Similarly, if one can identify a block of code or function that is executed repeatedly, the number of times that block is executed before the end is reached should also fall into a fixed range. One can use an automated system to infer loop invariants and bound the number of times the loop should execute [70].

Given a block of code and the WCET (either in units of time or the number of executions of that block of code), one can build a detector using a pair of hprobes. When one knows the wall clock time, one can insert one probe at the inside the block and another probe after the block. At the first probe, a timer is started (using the same technique as the heartbeat detector in Section 5.4.2). If the timer expires before the second probe (at the end) is reached, the detector reports a failure. If there is concern that the hypervisor or guest OS is over-provisioned and significant time sharing is taking place, one can use architectural invariants [28, 14] to only count the time when the application under consideration is being executed by monitoring context switch events using the CR3 register. For the case where a bound on the number of executions of the block of code is known, one can place one probe at the beginning of the loop and one immediately after the loop. If the probe inside the loop is executed more times than expected without the block being exited, then the detector can report failure (i.e., a range violation [71]). Depending on the needs of the user, the detector can either reset its state or remove itself when the exit probe is hit.

In addition to using the WCET, one can also observe the state of the system to detect an infinite loop [72]. When using KVM, the register state of the VM is saved in KVM data structures to be reloaded upon the next VM entry. As mentioned in Section 5.3.3, probes inserted by a kernel module in the host OS pass a structure describing the vCPU that generated the `int3` exception. This structure contains another structure with architecture specific information, including the register state at the time of the VM Exit. The detector can check this state at every loop iteration. If the registers remain constant across a large number of iterations, this static state can be attributed to an infinite loop in many applications.

In order to test the infinite loop detector, we used the same example as presented in Jolt [72]. That example is a bug found in a development branch of the Exuberant Ctags source code indexer.[7] In that bug, a string parsing loop would get stuck due to two variable names being transposed in the source code. The example input for the ctags indexer used in Jolt is the python scientific computing package numpy.[8] Specifically, the `_import_tools.py` file contains comments that are formatted in such a way that the bug is activated. In the fixed version of the code the loop executes only one iteration each of the twelve times it is entered, meaning a small threshold could also be used. Regardless of whether the threshold or register change method is used, this loop was easily detected in all experiments since it executes at a rate of thousands of times per second.

## 5.5 Performance

### 5.5.1 Methodology

All of our microbenchmarks and detector performance evaluations were conducted on a Dell PowerEdge R720 server with dual-socket Intel Xeon E5-2660 "Sandy Bridge" 2.20GHz CPUs (3.0 GHz turbo boost). To obtain runtime measurements, we have added an extra hypercall to KVM that starts and stops a timer inside the host OS. This allows us to obtain measurements independent of VM clock jitter. To ensure consistency among measurements, the test VMs were rebooted between each sample.

### 5.5.2 Microbenchmarks

We perform microbenchmarks that estimate the latency of a single hprobe, which is the time from when the VM executes `int3` until the VM is resumed (Steps 1–3 in Fig. 5.3). We run these microbenchmarks without a probe handler function to determine the lower bound of hprobe-based detector overhead. Since the round-trip latency of an individual VM Exit on Sandy Bridge CPUs has been estimated to take roughly 290 ns [73] and our hypercall measurement scheme induces additional VM Exits, it would be difficult to accurately measure the individual probe latency. Instead, we obtain

[7]http://ctags.sourceforge.net/
[8]http://www.numpy.org

a mean round-trip latency by repeatedly executing a probed function a large number of times (one million) and dividing by the total time taken for those executions. This helps remove jitter due to timer inaccuracies as well as the actual latency of the hypercall measurement system itself. For the test probe function we have added a no-op kernel module to the Guest OS that creates a dummy `noop` device with an `ioctl` that calls a `noop_func()` kernel function that performs no useful work (`return 0`). First, we insert an hprobe at the `noop_func()`'s location. Our microbenchmarking application starts by issuing a hypercall to start the timer and then an `ioctl` against the noop device. When the noop module in the guest OS receives the `ioctl`, it calls `noop_func()` one million times. Afterwards, another hypercall is issued from the benchmarking application to read the timer value.

For the microbenchmarking experiment, we used a 32bit Ubuntu 14.04 guest and measured 1000 samples. The mean latency (across samples) was found to be 2.6 $\mu$s. In addition to the Sandy Bridge CPU, we have also included data for an older generation 2.66GHz Xeon E5430 "Harpertown" processor (running the same kernel, KVM version, and VM image), which had a mean latency of 4.1 $\mu$s. The distribution of latencies for these experiments is shown in Fig. 5.7. The remainder of benchmarks presented use the Sandy Bridge E5-2660. The hprobe prototype requires multiple VM Exits per probe hit. However, in many practical cases the flexibility of dynamic monitoring and lower maintenance due to a simple implementation outweigh this cost. This flexibility can increase performance in many practical cases by allowing one to add and remove probes throughout the VM's lifetime, as will be demonstrated later. Furthermore, CPU manufacturers are constantly working to reduce the impact of VM Exits, as Intel's VT-x saw an 80% reduction in VM Exit latency over its first six years [73].

### 5.5.3 Detector performance

In addition to microbenchmarking individual probes, we measure the overhead of the example hprobe-based detectors presented in Section 5.4. All measurements in this section were obtained using the hypercall-based timer.

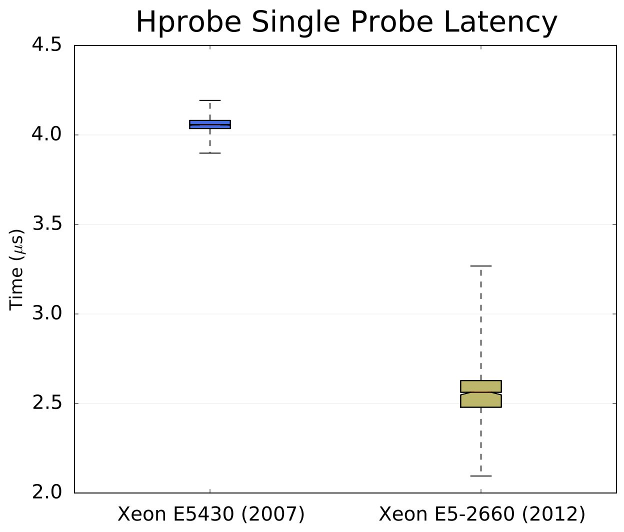


Figure 5.7: Single probe latency (parentheses are the CPU's release year). The E5-2660's larger range can be attributed to "Turbo Boost," where the clock scales from 2.2 to 3.0GHz. The shaded area is the quartile range (25th percentile to 75th percentile), whiskers are minimum/maximum, center is the mean, and notches in the middle are the 95% confidence interval of the mean.

### Emergency Exploit Detector

Our integer overflow detector that protects against the CVE-2008-0600 `vmsplice()` vulnerability is extremely lightweight. Unless `vmsplice()` is used, the overhead of the detector is zero since the probe will not be executed. The `vmsplice()` system call is rare (at least in open source repositories that we searched), so this zero overhead is overwhelmingly the common case. Keeping in mind that security vulnerabilities are often found in "cold" regions of code [74], we believe this low-overhead to extend beyond our simple example. One application that does use `vmsplice()` is Checkpoint/Restart in Userspace (CRIU).[9] CRIU uses `vmsplice()` to capture the state of open file descriptors referring to pipes. We used the Folding@Home molecular dynamics simulator [75] and the pi-qmc Monte Carlo simulator from earlier as test programs. We ran these applications in a 64-bit Ubuntu 14.04 VM. At each sample, we allowed the application to warm up (load input data and start the main simulation) and then checkpointed it. The timing hypercalls were inserted into CRIU to measure how long it takes to dump the application.

[9] http://www.criu.org/

Table 5.1: CVE-2008-0600 Detector w/CRIU

| **Application** | **Runtime (s)** | **95% CI (s)** | **overhead (%)** |
|---|---|---|---|
| F@H Normal | 0.221 | 0.00922 | 0 |
| F@H w/Detector | 0.228 | 0.0122 | 3.30 |
| F@H w/Naïve Detector | 0.253 | 0.00851 | 14.4 |
| pi-qmc Normal | 0.137 | 0.00635 | 0 |
| pi-qmc w/Detector | 0.140 | 0.00736 | 1.73 |
| pi-qmc w/Naïve Detector | 0.152 | 0.00513 | 11.1 |

This was repeated 100 times for each case with and without the detector and the results are tabulated in Table 5.1. From the table, we can see that there is a slight difference in the mean checkpoint time (roughly 3.3% for F@H and 1.7% for pi-qmc) and that the variance in the experiment with the detector active is higher for the Folding@Home case. When checkpointing Folding@Home, `sys_vmsplice()` was called 28 times, and 11 times for pi-qmc. We can attribute this to negative cache effects of the context switch when activating probes. We also measured another class of "Naïve" detector that probes the `system_call()` function (the entry point for all system calls) during the checkpoint as opposed to `sys_vmsplice()`. In the case where we probe on all system calls, we can see that there is a significant performance penalty (and the number of probe invocations increases to $\sim 3000$). We remind the reader that the detector only probes `sys_vmsplice()`, meaning the overhead incurred is only when taking a checkpoint.

### Application Heartbeat Detector

We use the pi-qmc simulator from Section 5.4.2 to measure the performance overhead of the application watchdog detector. The pi-qmc simulator allows configuration of its internal sampling and we utilize this feature to vary the length of the main loop. In order to determine how the detector impacts performance we measure the total runtime of each iteration of the main loop when the probe is inserted and run the program for 15 minutes. The results of our experiments are shown in Fig. 5.8.

From Fig. 5.8, we show that the detector does not affect performance in a statistically significant way. This is due to the fact that pi-qmc, like many scientific computing applications, does a large amount of work in each iteration of its main loop. However, by setting the threshold of the detector to a conservative value (like twice the mean runtime), one can achieve fault

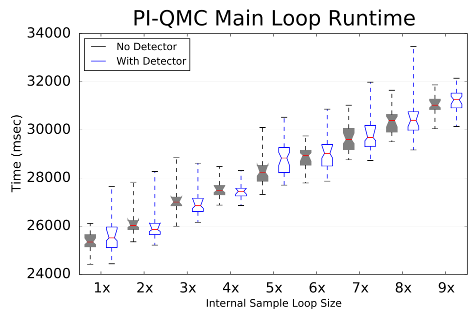


Figure 5.8: Benchmarking of the application watchdog detector. The horizontal axis indicates the scaling of an internal loop in the target pi-qmc program. The vertical axis shows a distribution of the completion time for each iteration of the main loop. The boxplot characteristics are the same as in Fig. 5.7.

detection in a far more acceptable timeframe than other methods like manual inspection. Furthermore, this detector goes beyond checking if the process is still running - it can detect any fault that causes a main loop iteration to halt (disk I/O hang, network outage when using MPI, software bug that does not lead to a crash, etc.).

### Infinite Loop Detector

In order to measure the performance overhead of our infinite loop detector, we use a patched version of the ctags application from Section 5.4.3. We ran ctags on the complete numpy source tree 60 times and obtained the mean completion time and 95% confidence interval. The results are tabulated in Table 5.2. There are two implementations of the detector used in these experiments, the "Naïve" detector and the "Smart" detector. The Naïve detector is the same detector as presented in Section 5.4.3 and the Smart detector has probes that dynamically add/remove themselves (i.e., the loop exit probe is only added after the loop is entered). When starting the application, the code segment containing the target function was paged out to disk (a clean boot for each sample). The rows in Table 5.2 with "Page fix" refer to the runs where we needed to use the EPT mechanism presented in Section 5.3.3. We also forced the application to page in the target code block at startup, represented by the "No Page Fix" samples. From Table 5.2 we can see that the performance impact of our solution to deal with paged-out user space

Table 5.2: Ctags on numpy source tree

| **Application** | **Runtime (s)** | **95% CI** (s) | **% overhead** |
|---|---|---|---|
| Normal | 1.13 | 0.0325 | N/A |
| Naïve ILD - Page Fix | 1.26 | 0.0229 | 11.5 |
| Naïve ILD - No Page fix | 1.26 | 0.0265 | 11.8 |
| Smart ILD - Page Fix | 1.14 | 0.0267 | 1.15 |
| Smart ILD - No Page Fix | 1.15 | 0.0215 | 1.9 |

application code is not statistically significant (compare the "Page Fix" rows of the same detector to the "No page fix" rows). However, using dynamic probes yields large performance gains. In the Naïve approach, the overall overhead is roughly 11.5% for this input data. With the Naïve detector, the first and second probes get executed 2585 and 54308, respectively. This is due to the fact that in many cases, the loop is skipped over, but the instruction immediately after the loop (i.e., what the second probe replaces) always gets executed. In the Smart approach, the first and second probe both get executed 2585 times (in correct operation on this input data, the loop has only one iteration), yielding a nominal difference between the Smart implementations and the base case without probes. If this loop had instead a high number of internal iterations, then one could use a similar dynamic probe approach, but retain the exit probe and remove the internal probe, adding it periodically or using a timeout mechanism. Note that the capability behind the "Smart" approach is unique to the dynamism in the hprobe framework.

## 5.6 Conclusions

The Hprobe framework is characterized by its simplicity, dynamism, and ability to perform application-level monitoring. Our prototype for this framework uses hardware-assisted virtualization and satisfies protection requirements presented in the literature [13]. We find that compared to past work, the simplicity with which detectors can be implemented and inserted/removed at runtime allows us to quickly develop monitoring solutions. Based on our experience, this framework is appropriate for use in real-world environments. From our sample detectors, we see that the framework is suitable for providing detection for bugs, random faults, and use as a stopgap measure against vulnerabilities.

# CHAPTER 6

# HSHIELD: MONITORING HYPERVISOR INTEGRITY

*This chapter introduces hShield, an architectural support for enforcing Control-Flow Integrity (CFI) of hypervisors.*

## 6.1 Introduction

HyperTap and Hprobes, introduced in the previous chapters, rely on the trustworthy of the underlying hypervisor to deploy their monitoring mechanisms. In this chapter, we validate this assumption and propose an approach to tighten the security of hypervisors. Particularly, we investigate VM-escape attacks, which are attacks that compromise hypervisor executions via the VM-hypervisor interface provided by hardware-assisted virtualization (HAV) (see Chapter 2 for a review of HAV). Based on the analysis of this threat model, we introduce a new monitoring technique that detects VM-escape attacks.

In a virtualized system, the hypervisor is a single-point-of-failure. It is the centralized component that manages interactions between VMs and the underlining physical resources, such as computing, networking, and storage. Most components in hypervisor are granted high-privilege to permit access to the shared resources. If one of those components is compromised, the entire virtualized system, including physical resources and other co-located VMs, is potentially compromised as well. When an attack works on one instance of hypervisor, the attack might be extending to affect other instances, which have the same version as the exploited hypervisor.

To detect VM-escape attacks, we introduce a monitoring framework called hShield. The core of hShield is an efficient control-flow integrity (CFI) enforcement, which is specifically designed based on our analysis of HAV-based hypervisors. In addition, our CFI method addresses two fundamental limitations of state-of-the-art CFI techniques [76, 77], namely imprecise control-flow graph (CFG) construction and the overhead of runtime CFI enforcement.

The design of hShield aims to provide the following features to hypervisor security monitoring:

**Resistance to VM escape attacks that subvert the control-flow of the hypervisor**. Many of attacks in this class can be classified into a zero-day attack – attackers exploit an undiscovered vulnerability in the implementation of a hypervisor, which allows them to execute malicious codes together with the normal execution of the hypervisor. hShield aims at detecting this class of attacks when they are being executed without knowing the vulnerability in advance.

**Negligible performance penalty in attack-free executions**. Similar to HyperTap and Hprobes, hShield employs the principle of event-driven monitoring, which is effective in detecting both transient and persistent attacks. Additionally, we analyzed the hypervisor execution model to extract events that hShield can efficiently monitor without incurring noticeable performance overhead when the system is in an attack-free state.

In order to evaluate hShield, we compared the result of our CFI technique with that of BinCFI [77], a state-of-the-art CFI implementation. Our experiments show that the CFG constructed using our method is more precise, thus, more secure in terms of CFI enforcement. More specifically, we showed that the approximation of BinCFI's static analysis leave dangerous paths in CFGs that can be exploited by attacks to perform a VM-escape. In addition, we showed that hShield can detect a real VM-escape attack that we crafted from a published vulnerability.

## 6.2 Assumptions and Threat Model

### 6.2.1 Assumptions

Our design targets at hypervisors that utilize hardware virtualization (e.g., Intel VT-x and AMD SVM) to manage VMs' executions. We make the following assumptions about the system.

*The underlining hardware virtualization is implemented correctly*, meaning that the only way to change from the VM privilege into the hypervisor priv-

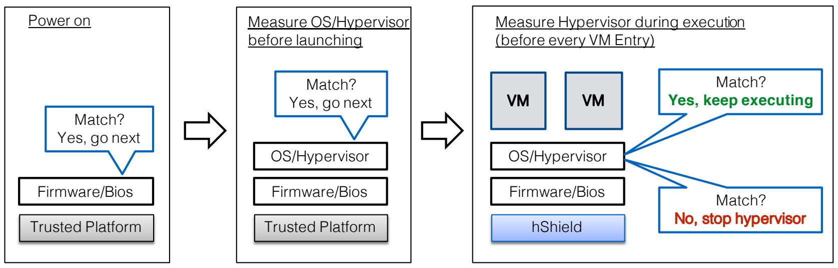


Figure 6.1: hShield protects hypervisor during execution. It assumes the integrity of the platform is guaranteed at load-time by a Trusted Platform, such as TPM or Intel TXT.

ilege is to going through the VM-exit interface, as described in Section 2.2. We do not handle attacks that exploit hardware vulnerabilities.

*The target host system is secured from physical tampering* (e.g., secured in a server room) and there is *no insider-attacker* (e.g., malicious administrators who already have remote access to the host system).

*The host system itself has limited direct open access from the outside world.* Preventing misuse of administrative credentials, e.g., through social engineering methods to illegally obtain an administrative credential and use it against the host system, is out of the scope of this work.

*The target host system is equipped with a trusted boot technology*, such as Trusted Platform Module (TPM) [78], or Intel Trusted eXecution Technology (TXT) [79], which ensures the integrity of the host system, including the hypervisor, at load-time. Note that we focus on ensuring the integrity of the hypervisor at runtime, given the integrity at load-time is guaranteed. Figure 6.1 shows how hShield works in tandem with trusted platform technologies.

## 6.2.2 Threat Model

Virtualization creates an isolated environment for each VM, so that multiple VMs can share common physical resources. The isolation is enforced so that a VM cannot access resources of host system, or other co-located VMs.

The primary threat model that we consider is classified as *VM escape attacks*. A VM escape attack is an attack that breaks the isolation wall created by hypervisor to allow programs running inside a VM to violate the integrity (i.e., alter the execution) of the hypervisor. In particular, an

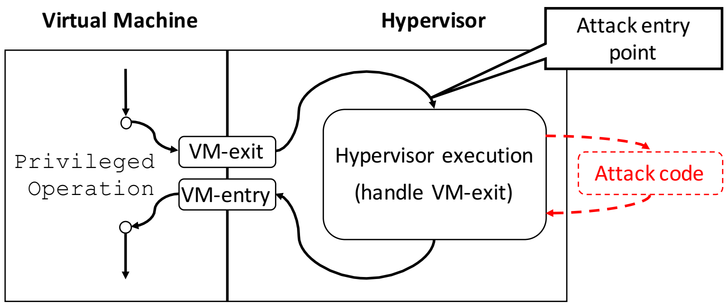


Figure 6.2: Illustration of a VM escape attack in a hardware virtualization-based hypervisor. The attack entry point is the interface the hypervisor created to handle VM-exit events. The attack diverts the execution of the hypervisor (represented by the red box) from the normal execution.

Table 6.1: Examples of high-profile VM escape-enabled CVEs from 2007-2015

| CVEs | Affected Hypervisors | Description |
|---|---|---|
| CVE-2007-1744 | VMware Workstation | Directory traversal vulnerability in Shared Folders feature |
| CVE-2008-0923 | VMWare ACE, Player, and Workstation | Path traversal vulnerability in VMwares shared folders implementation |
| CVE-2009-1244 | VMware Workstation, Player, ACE, Server, Fusion, and ESX | Cloudburst - virtual video adapter vulnerability |
| CVE-2012-0217 | Xen para-virtualization (PV) 64-bit | Vulnerability in system calls returning via sysret to a non-canonical RIP64-bit |
| CVE-2014-0983 | Oracle VirtualBox | 3D acceleration multiple memory corruption vulnerabilities |
| CVE-2015-(2336-2340) | VMare Workstation | Escaping VMware Workstation through COM1 (5 CVEs) |
| CVE-2015-3456 | QEMU, KVM, XEN | QEMU heap overflow flaw in floppy disk driver |
| CVE-2015-5154 | QEMU, KVM, XEN | QEMU heap overflow flaw while processing certain ATAPI commands |

attacker originally has full control over a VM. During the execution of the VM, the attacker is able to exploit unknown or unpatched vulnerabilities of the hypervisor software in attempt to compromise the hypervisor. The exploit allows the attacker to redirect control flow to execute malicious code. The malicious code can be either injected by the attacker or salvaged from existing code, e.g., through a return-oriented attack. The malicious code is executed at the privilege of the hypervisor, thus it has permissions to interfere and/or access secrets stored in the hypervisor and other co-located VMs. This is a powerful class of attack. Figure 6.2 demonstrates the VM escape attack via VM-exit interface.

Table 6.1 shows examples of VM escape-enabled vulnerability published in the CVE (Common Vulnerability Enumeration) database. These vulnerabilities share a common feature is that they allow exploit code running inside a VM to compromise the control follow integrity (CFI) of the underlining hypervisor.

The assumption about attackers having full control over a VM is based on practical settings of virtualized computing platforms. In a public IaaS environment, such as Amazon AWS EC2, Microsoft Azure, or IBM SmartCloud, users can create a VM to run custom software with very small cost. In other virtualized environments, in which users have no direct access to a VM, attackers may gain access to a VM through exploiting vulnerabilities in the VM's software (e.g., database or web service). Once having full control over a VM, an attacker can use the VM as an entry point to start attacking the underlining hypervisor.

We further break down VM escape attacks into transient attacks and permanent attacks. Transient attacks are attacks that occur stealthily fast in order to bypass periodic integrity measurements [80]. Meanwhile, permanent attacks once performed stay persistently in the target system. Majority of integrity measurement techniques are designed to cope with persistent attacks, leaving a gap for transient attacks to exploit [80]. Previous work [14, 15] has demonstrated the high effectiveness of transient attacks against periodic, or polling-based, monitoring. Our threat model includes both transient and permanent VM escape attacks.

## 6.3 hShield Approach Overview

This section describes the approach of our system, called hShield, to achieve the goals established in the previous section.

### 6.3.1 Limitations of Existing Control Flow Integrity Monitoring

CFI enforcement [76] is a common method used to prevent attacks relying on subverting executions of target systems (e.g., via exploiting buffer overflow vulnerabilities). In this method, valid execution paths of a program are represented as a Control-Flow Graph (CFG). The CFI runtime enforcement

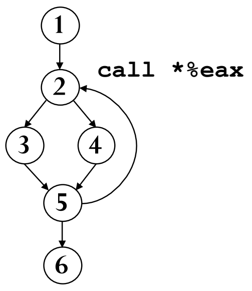


Figure 6.3: Example of a control flow graph (CFG). The nodes in the graph are basic blocks. The edges are flow control transfers from one basic block to another. The flow control transfer from node 2 to nodes 3 and 4 are indirect flow transfer, because the target of the `call` instruction is determined by the value of the register `eax` at runtime.

ensures that the target program must execute follow a valid path in a pre-determined CFG.

A CFG is a directed graph, in which a node represents a basic block[1] in the program, and a directed edge represents a transfer in the control-flow (e.g., a jump, call, or return instruction) from a source node, where the transfer is invoked, to the target node, where the transfer lands at. Figure 6.3 is an example of a CFG.

Runtime enforcing CFI aims at protecting target programs against unknown attacks based on the validity of CFG. A pre-determined CFG is essentially a whitelist of valid execution paths that are allowed to be executed. Hence, this whitelist-based monitoring approach can detect attacks that divert the target program to execute an invalid path according to the determined CFG, as opposed to a black-list-based monitoring approach which can only detect previously identified attacks.

The first challenge of CFI enforcement is to obtain a precise CFG of the target program. The existing approach to CFG construction is to use static analysis [76, 77] – analyzing the source code or binary of target programs. However, static analysis cannot determine indirect control flow transfer – the control-flow targets that are computed at runtime, e.g., function pointers or return addresses. In order to address this limitation, current CFI techniques employ approximation to statically determine such dynamic targets [77]. This imprecision is a potential source for attack to bypass CFI security run-

[1]A basic block is a consecutive sequence of instructions with no jump target except the entry and no jump source except the exit

time enforcement. For example, an attack can use a jump-to-libc attack to invoke functions that dynamically-incorrect, but statically-approximated.

The second challenge of CFI enforcement is to minimize the runtime overhead caused by runtime validation. The approach used by state-of-the-art CFI techniques is to perform target validation, e.g., validate whether the current jump follows a valid edge in the CFG, at the end of every basic block. The main challenge of this approach is to keep the performance overhead of the validation small due to the high frequency of basic block jumps.

### 6.3.2 hShield CFG Construction

hShield addresses the approximated CFG issue mentioned above by combining static analysis and profiling to construct a CFG. More specifically, we use static analysis to construct an initial CFG, which contains basic blocks (nodes in the CFG) and direct jumps (edges in the CFG), extracted from the target program binary. To derive indirect control flow information, we analyze the profiled traces of the target program execution under a set of representative workloads.

A trace records sequences of basic blocks visited during the execution of the target program. The order of basic blocks in a trace can be used to construct a CFG. For instance, two consecutive basic blocks B1 and B2 in a trace indicate that there is an edge from node B1 to node B2 in the CFG. A CFG constructed based on profiled traces contains both direct and indirect control flow information. However, the constructed CFG may not cover all possible valid paths that the target program may execute. The path coverage of the CFG is determined by the workloads used to execute and record the traces of the target program. All the collected traces are used to construct a CFG.

The initial CFG constructed using static analysis is merged with the CFG constructed based on profiled execution traces to produce a single CFG. That CFG contains both direct and indirect control flow information. This approach combines the advantages of both methods: static analysis can extract direct control flows, and execution traces contain indirect control flows which can only be accurately determined at runtime.

For the purpose of detecting VM escape attacks, the constructed CFG of

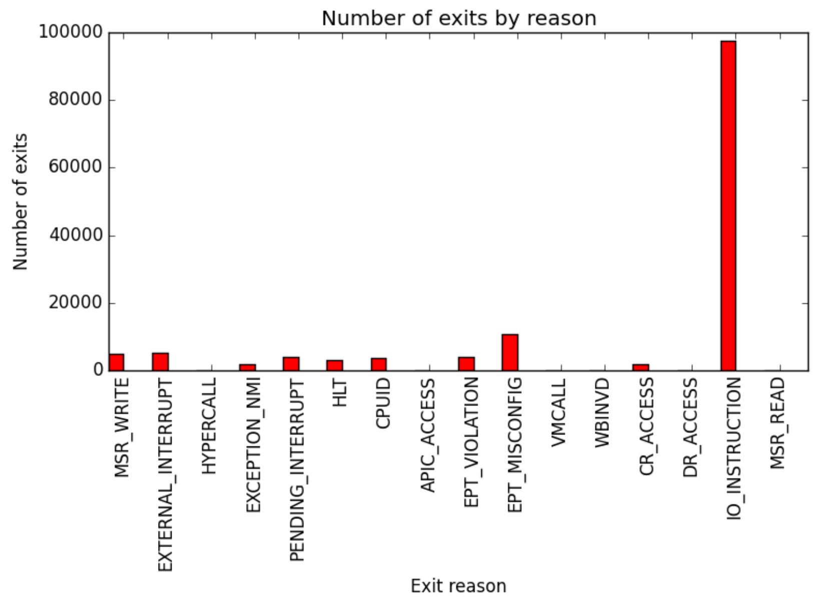


Figure 6.4: The distribution of VM Exit reasons profiled during the execution of a VM under CentOS Linux booting and UnixBench workloads.

a hypervisor needs to cover all valid execution paths from a VM Exit to the corresponding VM Entry. According to our threat model, this is the only attack vector that an attacker inside a VM can use to penetrate the hypervisor.

Figure 6.4 and 6.5 show the result of the CFG construction for the KVM-QEMU hypervisor. Figure 6.4 indicates that `IO_INSTRUCTION`s are the most frequent type of VM Exits: 82% of VM Exits triggered during the execution of a VM under CentOS booting and the set of utilities in the UnixBench benchmark are IO-related events.

Figure 6.5 shows the detailed CFG construction results for QEMU using various types of workloads. In a KVM-QEMU hypervisor, all IO-related VM Exits are handled by QEMU; thus, the collected events presented in the graph are IO-related events. The CFG was incrementally constructed using the traces collected by executing the workloads in order listed in the x-axis. Each of the workloads was run three times. The graph shows that neither new nodes nor edges were discovered after the PostMark benchmark, meaning that the CFG constructed by a subset of benchmarks is able to cover all paths to execute all the selected benchmarks.

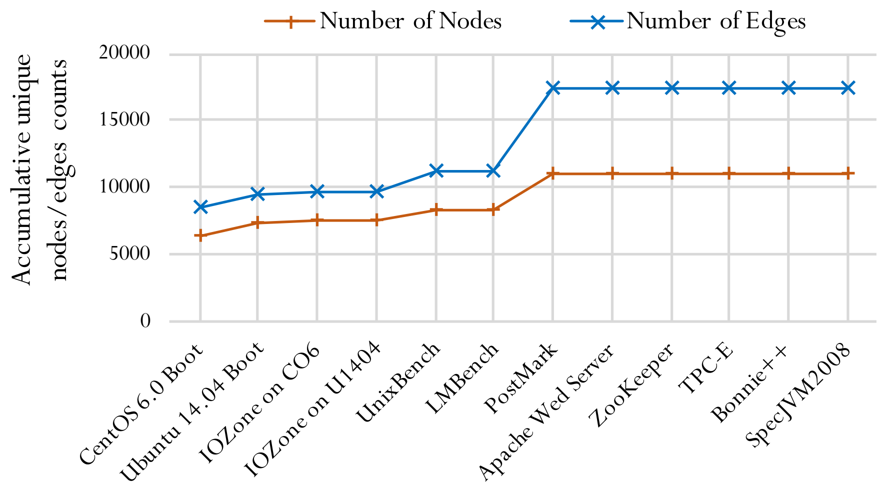


Figure 6.5: Profiling QEMU (IO and MMIO exits only) under different VM workloads.

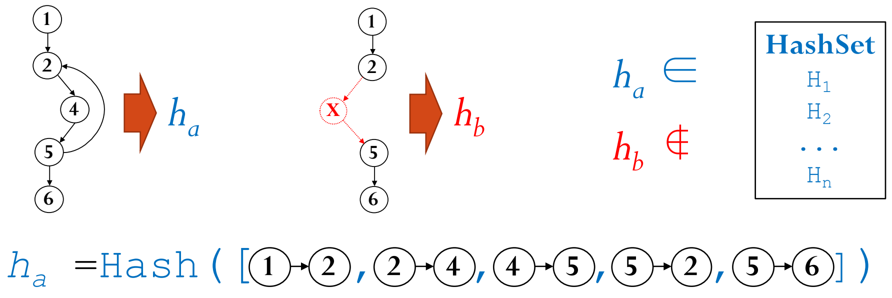


Figure 6.6: HashSet construction.

## 6.3.3 hShield Runtime Enforcement

hShield proposes a novel technique to improve the performance overhead of CFI runtime enforcement. This technique is particularly designed for HAV-based hypervisor execution model. Existing CFI enforcement performs validation at every control flow transfer. This validation is the major source of performance degradation incurring while executing protected programs. hShield's solution to this issue is to reduce the validation frequency by delaying it until a VM Entry is about to execute. Per our measurement, on average the frequency of executing a VM Entry is three orders of magnitude smaller than the frequency of a control flow transfer in the KVM-QEMU hypervisor.

hShield implements a hardware counter to compute a hashed value of hypervisor execution on-the-fly. Figure 6.6 describes how a hash is computed for each VM Exit handling. At the end of a VM Exit handling, triggered by a VM Entry event, hShield compares the computed hash against a pre-constructed HashSet. The pre-constructed HashSet represents the constructed CFG of the hypervisor. In other words, the HashSet is a whitelist of valid hypervisor execution paths. If an execution path is not listed in this whitelist, hShield flags it as an offended execution.

This approach of delaying CFI validation to the end of each VM Exit handling makes an important trade-off comparing the existing CFI enforcement: reducing performance overhead with the cost of longer detection latency. Since current techniques check for CFI at every control flow transfer, a CFI violation can be detected right before the execution of a malicious code. In hShield, the detection happens at the end of the violated VM Exit handling.

Section 6.4 details the hash function that hShield uses, and section 6.5 describes the architectural support to hShield.

## 6.4 Execution Hashing

The function of execution hashing is to map an arbitrarily long execution pattern input to a fixed length output hash value. An execution pattern is a stream of machine instructions executed by the processor.

### 6.4.1 Requirements

The hash function needs to be collision resistant. The this property is to ensure that it is computationally unfeasible to find a collision – an outside execution pattern that has the same hash as one of the white-listing members. Most standard cryptographic functions, such as MD5 or the SHA family, have this property.

The hardware implementation poses several extra constraints. First, the function needs to be *interactive*; that, is a hash can be continuously evaluated at runtime as input instructions coming, instead of storing the whole history of instructions and perform calculation at the end.

In addition, the hash function needs to facilitate the implementation of loop rerolling. hShield's loop rerolling involves frequent comparisons of basic blocks. Thus, hashing individual basic blocks should be an intermediate operation of the entire hashing scheme. Furthermore, loop rerolling requires re-evaluation of the final hashing output at runtime. For example, the hashing output changes when a loop iteration is removed. The ability to efficiently re-evaluate outputs at runtime is a necessity to enable hShield to cope with various issues, such as ones caused by hardware speculative executions. With speculative execution, a conditional branch may be predictively evaluated in advance, and unrolled and re-executed if the prediction was wrong.

### 6.4.2 Incremental collision-free hashing

The hashing function we select is a variation of the MuHASH function in the family of incremental collision-free hashing functions proposed in [81]. The key property which makes this family of hashing function suitable to our usage is that it is *incremental*. This property allows a hash value to be *updated* when a portion of the input is changed without caching or re-computing the value from scratch. We leverage this feature to facilitate loop rerolling implementation and cope with speculative execution.

This family of hashing function splits hashing into two phases: *randomize* and *combine*. Each input is broken into a sequence of blocks, and each block is *randomized* independently using a standard hashing function (e.g., a SHA function). The output of randomization is *combined* using an inexpensive commutative operation, e.g., modular multiplication in the case of MuHASH. Thanks to the communicative property of the combining operation, a hashed value can be updated by re-evaluating the randomized value of the modified input block.

Besides incrementality, MuHASH offers other properties that is suitable to hShield requirements:

- *Collision-resistance*: Based on an assumed-perfect standard hashing function (e.g., a SHA function), the security strength – the hardness of finding a collision – of the MuHASH is proven to be equivalent to the hardness of the discrete logarithm problem [81].

- *Parallel construction*: The randomization phase can be performed in parallel for each block. Note that property is stronger than *interactive* *construction*. We leverage this property to perform randomization per basic block with a small memory footprint.

- *Efficiency*: The construction uses only standard hashing function and inexpensive modular operation (as opposed to using exponentiation). The efficiency of this hashing function family is the same as using a standard hashing function on the entire input [81].

### 6.4.3 Runtime Construction

Essentially, the counter operates as a hash function $f$:

$$f : Exe \times \mathcal{S}alts \rightarrow \mathcal{R}ange.$$

The hash function $f$ maps from the space of finite variable-length instruction streams $Exe$ and a space of salt values $\mathcal{S}alts$ to the space of fixed length output value $\mathcal{R}ange$.

An execution $E \in Exe$ is a finite length stream of *basic block* $B_1B_2...Bn$, each basic block is a sequence of instructions $I_1I_2..I_m$. Each instruction $I_i$ is a valid x86 instruction represented in its binary form.

A salt $salt \in \mathcal{S}alts$ is a *unique* value for each system, thus it individualizes each system's counter table. A salt value is generated for a counter table when the profiling mode is executed. Note that for a salt to be effective, it does not need to be random. Thanks to the uniqueness property of salts, the work of crafting exploit code must be redone for each every system.

A hashing session starts on a VM-exit event, and ends on the corresponding VM-entry event. The continuous construction of the hash function during a session is as follows:

- Step 1: Session starts with resetting basic block counter to $i = 1$

- Step 2: For each incoming basic block $B_i$, concatenate a 32-bit binary encoding $\langle i \rangle$ of the basic block counter, and the *salt* value: $B'_i = \langle i \rangle.\langle salt \rangle.B_i$

- Step 3: (Randomization) Compute a hash value for the incoming basic block: $h_i = sha1(B'_i)$

- Step 4: (Combination) Combine $h_i$ using a combining operation $\odot$ to get the current hash value of the execution chunk:

$$f_i = \begin{cases} h_1, & i = 1 \\ f_{i-1} \odot h_i, & i > 1 \end{cases}$$

  As recommended by [81], we use the arithmetic operation multiplication modulo for combining operator to achieve collision-resistance.

- Step 5: Continue go back to step 2 until the session is ended.

Assuming that there are $n$ basic blocks in the evaluated execution chunk $E$, the final construction can be summarized in Figure 6.7, and as the equation follows:

$$f(E, salt) = \odot_{i=1}^{n} sha1(\langle i \rangle . \langle salt \rangle . B_i)$$

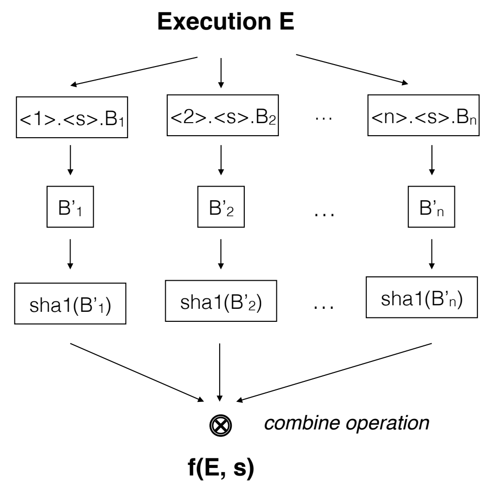


Figure 6.7: The construction of the incremental hashing function.

## 6.5 hShield Architectural Design

hShield is a security assisted hardware extension to the existing HAV to perform whitelist-based continuous monitoring of hypervisor executions. This section describes an example architectural design of hShield.

### 6.5.1 hShield Components

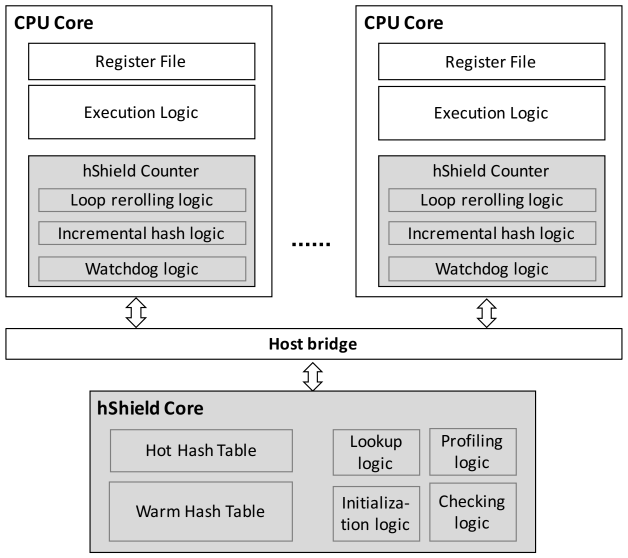


Figure 6.8: hShield architecture. Each CPU core has its own hShield counter to measure hypervisor execution at runtime. After a measurement is complete, the result is sent to the hShield core, which is a dedicated core per host system, to verify the measurement.

Each physical host is equipped with one hShield unit. An hShield unit consists of multiple per-core *hShield Counters* and one per-host *hShield Auditor*. Each hShield counter is built into a processor core, called the counter's *host core*. Each counter independently carries out the measurement of VM-exit handler executing on its host core. At the end of each measurement, the result, i.e., the hash represents the VM-exit handler execution, is sent to the auditor for whitelist member checking. The hShield auditor, implemented as a dedicated co-processor in this design, is responsible for securely loading and storing the whitelist, and efficiently executing whitelist updating and membership checking. Figure 6.8 illustrates this architecture.

hShield is designed to facilitate both whitelist construction and runtime checking. hShield auditor has two operational modes: *profiling* and *checking*. The profiling mode is used to support whitelist construction. In this mode, the auditor records hashes sent by counters to its hash tables. Meanwhile, the checking mode is used to validate hypervisor's executions during regular runs (e.g., with arbitrary clients' VMs). In this mode, the auditor validates an execution by comparing the hash sent by a counter against in the whitelist loaded in its hash tables.

hShield architectural design follows the separation of concerns principle. After being initialized by the centralized auditor, the operations of each counter are independent from each other, and also independent from the auditor. An hShield counter operates the same way whether the auditor is in the profiling or checking mode. The only component that stores the whitelist is the auditor. During runtime, there is only one type of unidirectional interaction between a counter and the auditor, which is sending-receiving a hash. There is no other interface that can leak information about the whitelist from the auditor to any of the processing cores.

Table 6.2 shows the interface of hShield Counters and hShield Auditor via the commands they process. The next subsections describe in details hShield counters and auditors.

Table 6.2: hShield Counter and Auditor commands

| Command | Callee* | Caller◇ | Mode+ | Parameters | Return |
|---|---|---|---|---|---|
| `HS_COUNTER_INIT` | Counter | Auditor | | () | `void` |
| `HS_WL_COUNT` | Auditor | Software | Profiling | () | Number of members |
| `HS_WL_READ` | Auditor | Software | Profiling | (`s`, `e`) | Whitelist members indexed from `s` to `e` |
| `HS_SALT_READ` | Auditor | Software | Profiling | () | *salt* |
| `HS_HASH` | Auditor | Counter | Profiling/ Checking | *hash* | `void` |

* **Callee** is the either a Counter or Auditor, which processes the commands.
◇ **Caller** is the component that can invoke the command. When a caller is "Software", that means this command is an instruction available for a software to use.
+ **Mode** is applicable for Auditor (as a callee) only. Mode specifies in which Auditor's mode ("Profiling", "Checking", or both) the command is available.

## 6.5.2 hShield Counters

Figure 6.9 depicts the finite state machine (FSM) of an hShield counter's operation. Each node of the FSM represents an operational state of a counter, and each edge represents an event that triggers a state transition. Note that the FSM can be terminated when it is in any state, and the "**End**" state is not shown in the figure for readability purposes. Besides the "**End**" state, an hShield counter can be in one of the following operational states:

"**Init**": At boot time, all hShield counters are initialized by the hShield auditor. Particularly, the hShield auditor instructs each of the hShield counters to load two common *salt* and *proof* values. When the initialization

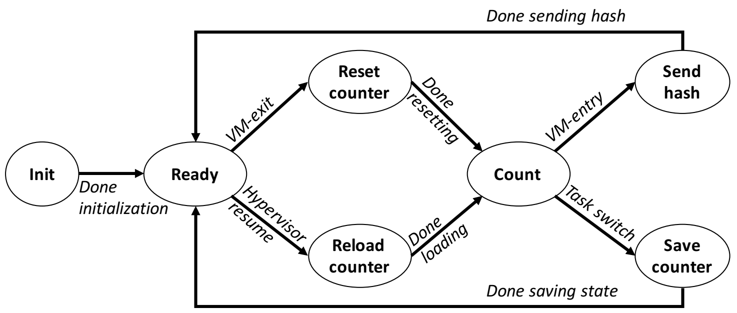


Figure 6.9: Finite state machine of an hShield counter operation. A node is a state of the counter, an edge is an event that triggers a state transition. All state can transit to the "**End**" state, which is not shown in this figure.

is done, represented by the "*Done initialization*" edge, the hShield counter transits to the "**Ready**" state.

"**Ready**": When an hShield counter is in this state, the processor is executing either in the guest mode (i.e., a VM is executing), or other tasks that do not belong to the hypervisor. Upon a "*VM-exit*" event, the counter transits to the "*Reset counter*" state. Meanwhile, upon an event that indicates "*Hypervisor resumed*" (e.g., a task switch event that the to-be-executed task belongs to the hypervisor), the counter transits to the "*Reload counter*" state.

"**Reset counter**": An hShield counter in this state is to respond to a VM-exit event issued by its host core. In this state, the counter resets all its internal state, e.g., the basic block counter, to get ready for a new hashing session. Upon completing the resetting, the counter transits to the "**Count**" state.

"**Reload counter**": In this state, the hShield counter loads an on-going hashing session context from memory to its internal state. The counter only loads the context which was properly signed using its hShield Proof. Upon completing the loading, the counter transits to the "**Counter**" state.

"**Count**": When an hShield counter is in this state, the host core is executing a hypervisor task that handles a VM-exit. In this state, the counter executes a hashing session, which implements the execution inference techniques and incremental hashing scheme (detailed in Section 6.4). In the event of a task switching, the counter suspends the on-going hashing session, and then moves to the "**Save counter**" state. In the event of an VM-entry, which signifies the end of the on-going hashing session, the counter computes the final hash of the hypervisor execution, and then transits to the "**Send Hash**" state.

"**Save counter**": In this state, the hShield counters save the context of the on-going hashing session to main memory. The saved data is signed with the hShield Proof to prevent tampering. Upon completing the saving, the counter transits back to the "**Ready**" state.

"**Send Hash**": This state marks the end of a hashing session by sending its result to the hShield auditor. Upon completing the sending, the counter transits back to the "**Ready**" state.

The next subsections discuss some issues in implementing hShield counters. Since an implementation of hShield is platform-dependent, we often concertize our discussion by using examples from Intel VT-x in the x86 architecture.

### HAV Integration

The implementation of hShield counters requires the ability to intercept VM-exit and VM-entry events. In Intel VT-x, in response to a VM-exit, the processor performs a sequence of operations to save guest state and then load host state. To incorporate an hShield counter, one would need to add the operation of the state "**Reset counter**" to the end of existing sequence of operations. Similarly, in response to a VM-entry (e.g., triggered by either of the instructions `VMLAUNCH` or `VMRESUME` in Intel VT-x), we would need to add the operation of the state "**Send Hash**" to the beginning of the VM-entry operations.

### Handling Interrupts

hShield counters exclude the executions of interrupt and exception handlers in the construction of execution hashes. In order to do so, we can leverage the existing hardware interrupt signal handling mechanism. When a processor detects the occurrence of an interrupt signal, it suspends the currently running task, saves the task context, and then starts executing the interrupt handler specified in the Interrupt Descriptor Table (IDT). When the interrupt handler is complete (e.g., upon an invocation of an `IRET` instruction), the processor resumes the execution of the suspended task to ensure program

Table 6.3: Content of an hShield Counter session context

| **Field** | **Size** | **Notes** |
|---|---|---|
| Current SHA1 hash | 20B | State to resume SHA1 computation |
| Instruction cache | 64B | of the current basic block |
| Basic block counter | 8B | Number of basic blocks |
| Current MuHash hash | 20B | Intermediate result of MuHash |
| *Checksum** | 20B | Checksum of the context signed by hShield Proof |
| **Total size** | **132B** | |

* *Checksum* is not a field in a session context, it is computed and saved together with a session context in memory.

continuity. Similarly, hShield pauses its on-going hashing session when an interrupt occurs, and then resumes the session when the interrupt handler is complete.

### Handling Task Switching

On a physical core, a running VM-exit handling task, the subject of hShield counter measurement, can be scheduled out to yield CPU to other tasks. Therefore, the context of the on-going hashing session associated with the hypervisor task needs to be saved together with the context of the hypervisor task, so that the hashing session can be resumed when the hypervisor task is resumed. This situation is handled by the "**Save counter**" state.

The context of an hShield counter to be saved is described in Table 6.3. A *checksum* of the session content signed by the hShield Proof is stored together with the session content. This checksum is used by hShield counter when resuming the hashing session to verify the integrity of the session context. This mechanism is to defeat false key injection attacks while a session context is stored in memory.

### Handling Multi-cores

HAV uses the notion of virtual CPU (VCPU) cores to indicate a VM's CPU core. Each physical CPU core can handle one VCPU core at the same time, similar to tasks. And VM-exits are generated and handled independently on each physical CPU core. Therefore, each physical core needs to maintain an independent hashing session, which can run in parallel with other hashing sessions happening on other physical cores.

In order to handle concurrent hashing sessions, we tailor an hShield counter to each CPU core. This approach is similar to that of existing performance counters. When a VM-exit is trapped in a physical core, the hashing session on that core is started to measure the execution of the VM-exit handler running on that core.

### 6.5.3 hShield Auditor

An hShield Auditor is a centralized component that manages the whitelist for a host system. An hShield Auditor operates in either of the two modes: *profiling* and *checking*. Setting which mode hShield Auditor operates on is done through the BIOS.

#### Profiling Mode

The profiling mode is used to facilitate the construction of the target hypervisor whitelist. This mode is also considered the *unsafe* mode of hShield Auditor, because its whitelist can be read and updated. Thus, the profiling mode must be ran in a strictly controlled environment with known-good VM workloads. In this mode, an hShield Auditor performs the following tasks:

**At boot time**, the following tasks are performed in a sequence:

1. Generates a new *salt* value.
2. Generates a new $proof$ value.
3. Broadcasts the `HS_COUNTER_INIT` command together with the *salt* and $proof$ values to all hShield Counters in the host to trigger their initialization process.

**During runtime**, the following tasks are performed in response to specific events:

- Upon receiving a hash from a Counter, the Auditor updates its hashing tables. The updating process is described in Section 6.5.3.
- Upon receiving a `HS_WL_COUNT` instruction, the Auditor returns the number of whitelist members.

- Upon receiving a `HS_WL_READ` instruction, the Auditor returns the hash corresponding to the specified whitelist member.
- Upon receiving a `HS_SALT_READ` instruction, the Auditor returns the value of the generated *salt*.

The `HS_WL_READ` and `HS_SALT_READ` instructions are used at the end of the profiling process to fetch the whitelist from the hShield Auditor to persist to the host's storage.

### Checking Mode

The checking mode is used for runtime monitoring of the target hypervisor, given that the whitelist has been properly constructed. In this mode, an hShield Auditor performs the following tasks:

**At boot time**, after the integrity of the host system is verified, e.g., by TPM and Intel TXT, the following task are performed in a sequence:

1. Load the whitelist and *salt* from the host persistent storage.
2. Generates a new $proof$ value.
3. Broadcasts the `HS_COUNTER_INIT` command together with the *salt* and $proof$ values to all hShield Counters in the host to trigger their initialization process.

**During runtime**:

- Upon receiving a hash from a Counter, the Auditor verify the membership of the hash.

Regardless of the hShield Auditor's operational mode, the operation of the hShield Counters in the same host is not affected: upon each VM-entry, the corresponding hShield Counter send a hash to the centralized Auditor.

### Hash Tables

Hash tables are hShield Counters internal storage to keep the whitelist. An hShield Counter contains two hash tables: *hot* and *warm*. The two tables function in the same way, except the following differences: The hot table's size is smaller than the warm table's; the hot table stores the top popular whitelist members, while the warm table stores the less popular whitelist members; a membership check operation is performed in the hot table first, and if there is no hit in the hot table, the operation is then performed in the warm table.

The hot-warm hash table design is to take advantage of the observed distribution of the frequency of the hit rate of whitelist members. The hot table is smaller, but stores the most frequently hit whitelist members.

## 6.6 Evaluation

### 6.6.1 CFG Constructed by Dynamic Profiling

To understand the feasibility of constructing the whitelist, some sample data was collected using the x86 performance counters [82]. The data was collected using a CentOS 7 virtual machine with 512 MB of RAM. The workloads used to collect data included the iozone file-system benchmark and the ApacheBench server performance measuring tool. The hardware performance counters instrumented were the total instructions retired and the total number of branch instructions.

These workloads generated two sets of data, one for QEMU and one for KVM. The data is in the form of a vector counters combined with a single exit reason, which describes the context of the measurement collection. In the QEMU data set, exit reasons observed were limited to memory mapped and traditional input/output. This is likely because an exit into QEMU mostly occurs to access hardware-related features (because QEMU is a machine emulator). The KVM data set contains a large number of exit reasons, which allow for more detailed analysis of the underlying workload. The reason for the higher level of detail in KVM's exit reasons is that QEMU must trap to KVM for each privileged instruction, and this occurs in a wide range of scenarios.

Examining a graph of new signatures over time allows us to observe new workloads visually. A new workload (or one that is malicious) will cause the graph to slope upwards. We expect to see a flat trend for workloads that have already been added to the whitelist. Breaking down these graphs by individual exit type also helps expose certain relationships and properties about workloads. We also expect to see a large number of repeated signatures in the data which will assist in keeping the whitelist memory usage reasonable.

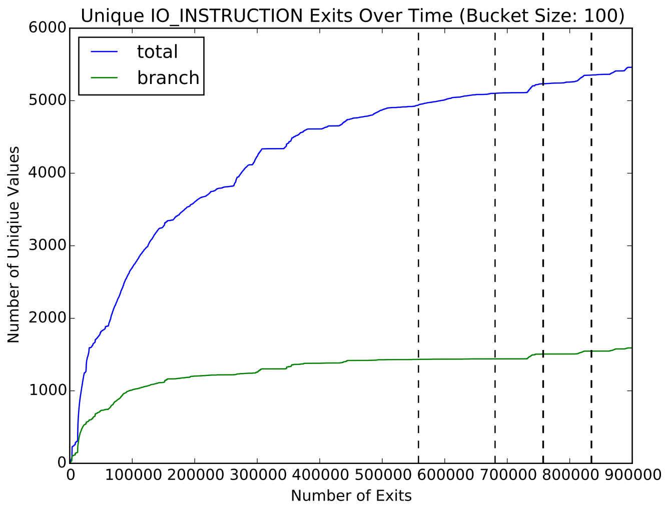


Figure 6.10: The number of unique IO exits observed over time from the KVM portion of the data. The vertical lines, from left to right, represent when the shell is first displayed, the beginning of the first iozone run, the end of the first run (and beginning of the second), the end of the second run (and beginning of the third), and the end of the third run as the end of the graph.

We designed a simple experiment to test this hypothesis. To collect data, the iozone filesystem benchmark was installed on the VM. The VM was then started and iozone was run three consecutive times. The data was then grouped into buckets. A bucket is defined as a range of values of a given size. For example, if the bucket size is 10, the first bucket contains all values 0-9, the second contains 10-19, etc. The bucket size can be tweaked to reveal different insights about the data. Figure 6.10 shows the results of graphing this data. The graph shows that during an iozone run, there are new values observed outside of this threshold. This shows the feasibility of hShield's ability to detect new workloads.

The data collected shows some other interesting properties. Only 337621 (37.51%) of the 900000 signatures collected were unique (this value was obtained without using any grouping by bucket). Assuming that we use a 20-byte cryptographic hashing function to represent an execution path, we would need about 6.4 Mbyte of storage to construct the whitelist. We also observed that the popularity of execution paths follows the zip distribution. We can leverage this property to further partition the set of execution paths according to the measured popularity to favor the lookup time for more frequently used execution paths.

## 6.6.2 Detection of Real Attacks

In this section, we describe a VM-escape attack that we implemented based on the QEMU's Venon vulnerability discovered in May 2015 [21]. Venom is a buffer-overflow vulnerability in the implementation of the QEMU floppy disk driver's buffer. This vulnerability allows any guest VM to overwrite the heap memory of the host QEMU process by executing an `io_out` instruction.

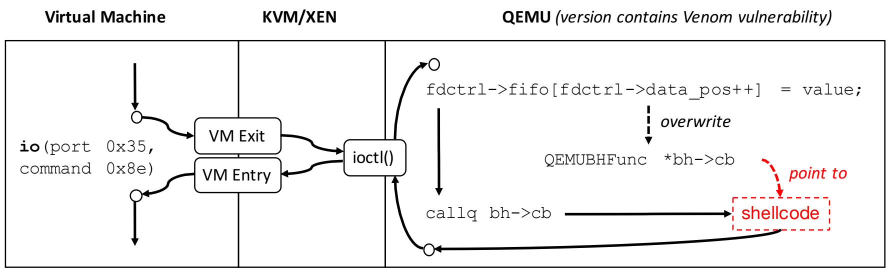

Virtual Machine
KVM/XEN
QEMU (version contains Venom vulnerability)
io(port 0x35, command 0x8e)
VM Exit
VM Entry
ioctl()
fdctrl->fifo[fdctrl->data_pos++] = value;
overwrite
QEMUBHFunc *bh->cb
point to
callq bh->cb
shellcode

Figure 6.11: Exploit Venom vulnerability for a VM escape attack.

By examining the QEMU's heap memory layout, we discover that one can use this vulnerability to overwrite a function pointer stored in the heap to redirect QEMU to execute a function of their choice. More specifically, the exploitable function pointer is in the `QEMUBHFunc` structure, which determines how to handle a bottom-half (BH) routine of an interrupt. Moreover, this function pointer allows the caller to pass an arbitrary parameter to the target function. This parameter is also stored in the `QEMUBHFunc`, which can be controlled by the attacker. Our exploit code is able to overwrite the content of this function pointer as well as the to-be-passed parameter to direct QEMU to execute the `system()` function – a libc function that runs a given shell command. The shell command could be any command, such as starting a backdoor program, or disable system firewall, as long as the QEMU process has permission to run. Figure 6.11 illustrates is exploit in a KVM-QEMU hypervisor setup.

In our experiment, hShield was able to identify that the execution of the hypervisor to handle the VM-exit initiated by the exploited `io_out` instruction from the malicious VM. Our experiment covered two scenarios. The first scenario tested the detection using a CFG constructed without using a floppy disk workload. Thus, all execution paths of QEMU that go to the floppy driver are invalid according to this CFG. The second scenario tested the detection using a CFG constructed by using a floppy disk workload. The detection is raised when the function pointer in the `QEMUBHFunc` is invoked, because in our CFG, the `system()` in libc is not a valid target of the call site of this invocation.

## 6.7 Conclusion

This chapter presents hShield together with a new CFI enforcement method, both of which are specifically designed to detect VM-escape attacks. We have shown the advantages of our proposed CFI enforcement method, e.g., more precise CFG and lower computational overhead for runtime CFI enforcement, over state-of-the-art CFI enforcement techniques. Our hShield system is able to detect hand-crafted VM-escape attacks that are based on published vulnerabilities.

hShield is the lowest-level system in our monitoring series, including Hprobes, HyperTap, and hShield. The three monitoring systems create a complete bottom-top chain-of-trust for the entire virtualization software stack.

# CHAPTER 7

# CONCLUSION

This thesis describes three contributions of our research toward achieving resiliency for virtualized computer systems. We took a multi-layered monitoring approach, in which we built specific systems that provide monitoring capability at specific layers of the virtualization software stack, namely, HyperTap to protect guest OS, Hprobes to protect guest applications, and hShield to protect the hypervisor.

First, we present HyperTap, an out-of-VM monitoring framework that leverages existing VM-exit mechanism of HAV to provide low-cost event-driven monitoring. In HyperTap, we define a rich set of rules that can be used to robustly derive information about guest OS activities from hardware state. We then demonstrate use cases in which this information is used to build reliability and security detectors that cannot be bypassed.

Next, we describe another out-of-VM monitoring framework, called Hprobes, which focus on providing dynamic monitoring placement. Hprobes addresses the main limitation of HyperTap, which is the inflexibility in adding and removing monitoring hooks at runtime. In addition, Hprobes's dynamic hook placement mechanism facilitates the construction of out-of-VM guest applications monitoring. HyperTap and Hprobes can work in tandem to provide a complete monitoring coverage to both guest OS and applications.

Finally, we introduce hShield to continuously monitor the integrity of hypervisor executions. hShield is embodied with a novel CFI technique which is designed to detect VM-escape attacks. hShield completes the chain-of-trust in our series of multi-layered monitoring systems.

While our monitoring systems are specifically designed and customized for HAV-based virtualization systems, we believe that many fundamental principles what we explored in this thesis can be applied to security and reliability monitoring in general. Throughout the thesis, we argumentatively and experimentally show that the polling-and-scanning monitoring paradigm must be avoided in security monitoring. Instead, we strongly advocate the use of continuous monitoring, specifically, event-driven monitoring. Our systems demonstrate that continuous monitoring can be strategically designed to achieve low-cost, yet high-detection, coverage.

# REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” vol. 37, pp. 29–43, 2003.

[2] 451Research, “Theinfopro servers and virtualization study,” 2013. [Online]. Available: https://451research.com/report-long?icid=2169&task=download&file=summary

[3] A. Gillen, M. Eastwood, I. Feng, K. Stolarski, J. Scaramella, and G. Chen, “Worldwide virtual machine 2013-2017 forecast: Virtualization buildout continues strong,” IDC Report, 2013.

[4] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developers Manual Volume 3 (3A, 3B & 3C): System Programming Guide*, September 2014.

[5] Advanced Micro Devices Inc, *AMD64 Architecture Programmers Manual Volume 2: System Programming*, May 2013.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “KVM: The Linux virtual machine monitor,” in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.

[8] T. Garfinkel and M. Rosenblum, “A virtual machine introspection based architecture for intrusion detection,” in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[9] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection and monitoring through VMM-based out-of-the-box semantic view reconstruction,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, pp. 12:1–12:28, Mar 2010.

[10] B. D. Payne, M. de Carbone, and W. Lee, “Secure and flexible monitoring of virtual machines,” in *Proceedings 23rd Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2007, pp. 385–397.

[11] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP).* IEEE, 2011, pp. 297–312.

[12] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 279–290.

[13] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP 2008).* IEEE, 2008, pp. 233–247.

[14] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. Iyer, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, June 2014, pp. 13–24.

[15] G. Wang, Z. J. Estrada, C. Pham, Z. Kalbarczyk, and R. K. Iyer, "Hypervisor introspection: A technique for evading passive virtual machine monitoring," in *the 9th USENIX Workshop on Offensive Technologies (WOOT 15).* Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/wang

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

[17] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *Proceedings of the 18th USENIX Security Symposium*, 2009, pp. 383–398.

[18] P. Cao, E. Badger, Z. Kalbarczyk, R. Iyer, and A. Slagell, "Preemptive intrusion detection: Theoretical framework and real-world measurements." ACM, 2015, p. 5.

[19] T. R. Flo, "Ninja: Privilege escalation detection system for gnu/linux," Ubuntu Manual, http://manpages.ubuntu.com/manpages/lucid/man8/ninja.8.html, 2005.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra et al., “The worst-case execution-time problemoverview of methods and survey of tools,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[21] NIST, “Vulnerability summary for cve-2015-3456,” USA, 2015. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3456

[22] S. Garfinkel, *Architects of the Information Society: 35 Years of the Laboratory for Computer Science at MIT.* Mit Press, 1999.

[23] Spiceworks, “Start of smb it report,” Spiceworks Report, http://www.spiceworks.com/marketing/state-of-smb-it, 2014.

[24] A. Bartels, J. R. Rymer, J. Staten, K. Kark, J. Clark, and D. Whittaker, “The public cloud market is now in hypergrowth: Sizing the public cloud market, 2014 to 2020,” Forrester Report, https://www.forrester.com/The+Public+Cloud+Market+Is+Now+In+Hypergrowth/fulltext/-/E-RES113365?intcmp=blog:forrlink, 2014.

[25] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” p. 121, 1973.

[26] N. Bhatia, “Performance evaluation of intel ept hardware assist,” *VMware, Inc*, 2009.

[27] Y. Fu and Z. Lin, “Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection,” in *Security and Privacy (SP), 2012 IEEE Symposium on.* IEEE, 2012, pp. 586–600.

[28] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Antfarm: Tracking processes in a virtual machine environment,” in *Proceedings of the USENIX Annual Technical Conference*, 2006, pp. 1–14.

[29] 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 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’08. New York, NY, USA: ACM, 2008, pp. 91–100.

[30] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, “Secure in-vm monitoring using hardware virtualization,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS ’09. New York, NY, USA: ACM, 2009, pp. 477–487.

[31] Q. Liu, C. Weng, M. Li, and Y. Luo, "An in-vm measuring framework for increasing virtual machine security in clouds," *Security & Privacy, IEEE*, vol. 8, no. 6, pp. 56–62, 2010.

[32] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 839–850.

[33] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 51–62.

[34] J. Pfoh, C. Schneider, and C. Eckert, "Nitro: Hardware-based system call tracing for virtual machines," in *Advances in Information and Computer Security.* Springer, 2011, pp. 96–112.

[35] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 416–427.

[36] Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer, "Dynamic vm dependability monitoring using hypervisor probes," in *European Dependable Computing Conference (EDCC)*, 2015.

[37] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 103–115.

[38] Nergal, "The advanced return-into-lib(c) exploits: Pax case study," *Phrack #58, Article 4, http://www.phrack.org/issues.html?issue=58&id=4*, 2001.

[39] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "Spectre: A dependable introspection framework via system management mode," in *Proceedings of The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, June 2013.

[40] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan, "Vigilant–out-of-band detection of failures in virtual machines," *Operating systems review*, vol. 42, no. 1, p. 26, 2008.

[41] M. Bishop, "A model of security monitoring," in *Fifth Annual Computer Security Applications Conference.* IEEE, 1989, pp. 46–52.

[42] 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*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 28–37.

[43] L. Wang, Z. Kalbarczyk, W. Gu, and R. K. Iyer, "An os-level framework for providing application-aware reliability," in *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on.* IEEE, 2006, pp. 55–62.

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

[45] 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 (ARES).* IEEE, 2009, pp. 74–81.

[46] K. S. Yim, Z. T. Kalbarczyk, and R. K. Iyer, "Quantitative analysis of long-latency failures in system software," in *Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on.* IEEE, 2009, pp. 23–30.

[47] D. Cotroneo, R. Natella, and S. Russo, "Assessment and improvement of hang detection in the linux operating system," in *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on.* IEEE, 2009, pp. 288–294.

[48] J. Butler and G. Hoglund, "Vice–catch the hookers," *Black Hat USA*, vol. 61, 2004.

[49] D. Sd, "Linux on-the-fly kernel patching without lkm," *Phrack Magazine #58, Article 7, http://www.phrack.org/issues.html?id=7&issue=58*, 2001.

[50] T. Ormandy, "The gnu c library dynamic linker expands $origin in setuid library search path," 2010, [Online; accessed 9-April-2013]. [Online]. Available: http://seclists.org/fulldisclosure/2010/Oct/257

[51] SecurityFocus, "Linux kernel cve-2013-1763 local privilege escalation vulnerability," 2013, [Online; accessed 15-Nov-2016]. [Online]. Available: http://www.securityfocus.com/bid/58137

[52] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Security and Privacy (SP), 2012 IEEE Symposium on*, 2012, pp. 143–157.

[53] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proceedings of the Network and Distributed Systems Security Symposium*, vol. 33, 2003.

[54] N. Provos, "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, vol. 1, no. 8. Washington, DC, 2003, p. 10.

[55] A. P. Kosoresow and S. Hofmeyer, "Intrusion detection via system call traces," *IEEE Software*, vol. 14, no. 5, pp. 35–42, 1997.

[56] J. Criswell, N. Geoffray, and V. S. Adve, "Memory safety for low-level software/hardware interactions." in *Proceedings of the 2009 USENIX Security Symposium*, 2009, pp. 83–100.

[57] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 351–366.

[58] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2011.

[59] P. Padala, "Playing with ptrace, part1," *Linux Journal*, no. 103, Nov. 2002. [Online]. Available: http://www.linuxjournal.com/article/6100

[60] R. Krishnakumar, "Kernel korner: kprobes-a kernel debugger," *Linux Journal*, vol. 2005, no. 133, p. 11, 2005.

[61] W. Feng, V. Vishwanath, J. Leigh, and M. Gardner, "High-fidelity monitoring in virtual computing environments," in *Proceedings of the International Conference on the Virtual Computing Initiative*, 2007.

[62] B. D. Payne, "Simplifying virtual machine introspection using libvmi," *Sandia Report*, 2012.

[63] NIST, "Vulnerability summary for cve-2008-0600," USA, 2008. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0600

[64] J. Corbet, "vmsplice(): the making of a local root exploit," 2008, [Online; accessed 15-Nov-2016]. [Online]. Available: http://lwn.net/Articles/268783/

[65] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems.* ACM, 2009, pp. 187–198.

[66] S. J. Vaughan-Nichols, "No reboot patching comes to linux 4.0," 2015, [Online; accessed 15-Nov-2016]. [Online]. Available: http://www.zdnet.com/article/no-reboot-patching-comes-to-linux-4-0/

[67] D. P. Bovet and M. Cesati, *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[68] D. Spinellis, "Trace: A tool for logging operating system call transactions," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 4, pp. 56–63, 1994.

[69] M. Gilbert and J. Shumway, "Probing quantum coherent states in bilayer graphene," *Journal of computational electronics*, vol. 8, no. 2, pp. 51–59, 2009.

[70] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[71] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer, "Automated derivation of application-specific error detectors using dynamic analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 640–655, 2011.

[72] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *ECOOP 2011–Object-Oriented Programming.* Springer, 2011, pp. 609–633.

[73] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software techniques for avoiding hardware virtualization exits." in *USENIX Annual Technical Conference*, 2012, pp. 373–385.

[74] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *the 36th IEEE Symposium on Security and Privacy*, no. EPFL-CONF-205055, 2015.

[75] S. M. Larson, C. D. Snow, M. Shirts et al., "Folding@ home and genome@ home: Using distributed computing to tackle previously intractable problems in computational biology," 2002.

[76] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.

[77] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13).* Washington, D.C.: USENIX, 2013, pp. 337–352.

[78] T. C. Group, “Trusted computing group: Trusted platform module,” 2015, [Online; accessed 15-Nov-2016]. [Online]. Available: http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/

[79] I. Corporation, “Trusted compute pools with intel(r) trusted execution technology,” 2015. [Online]. Available: http://www.intel.com/txt

[80] 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*, ser. CCS ’10. New York, NY, USA: ACM, 2010, pp. 38–49.

[81] M. Bellare and D. Micciancio, “A new paradigm for collision-free hashing: Incrementality at reduced cost,” in *Advances in CryptologyEUROCRYPT97*. Springer, 1997, pp. 163–192.

[82] V. Weaver, D. Terpstra, and S. Moore, “Non-determinism and overcount on modern hardware performance counter implementations,” in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 215–224.