DYNAMIC RELIABILITY AND SECURITY MONITORING: A VIRTUAL MACHINE APPROACH

BY

ZACHARY J. ESTRADA

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

Doctoral Committee:

Professor Ravishankar K. Iyer, Chair
Research Professor Zbigniew T. Kalbarczyk
Associate Professor Michael D. Bailey
Professor Roy H. Campbell
Professor Wen-Mei W. Hwu

Abstract

While one always works to prevent attacks and failures, they are inevitable and situational awareness is key to taking appropriate action. Monitoring plays an integral role in ensuring reliability and security of computing systems. Infrastructure as a Service (IaaS) clouds significantly lower the barrier for obtaining scalable computing resources and allow users to focus on what is important to them. Can a similar service be offered to provide on-demand reliability and security monitoring?

Cloud computing systems are typically built using virtual machines (VMs). VM monitoring takes advantage of this and uses the hypervisor that runs VMs for robust reliability and security monitoring. The hypervisor provides an environment that is isolated from failures and attacks inside customers' VMs. Furthermore, as a low-level manager of computing resources, the hypervisor has full access to the infrastructure running above it. Hypervisor-based VM monitoring leverages that information to observe the VMs for failures and attacks. However, existing VM monitoring techniques fall short of "as-a-service" expectations because they require *a priori* VM modifications and require human interaction to obtain necessary information about the underlying guest system. The research presented in this dissertation closes those gaps by providing a flexible VM monitoring framework and automated analysis to support that framework.

We have developed and tested a dynamic VM monitoring framework called Hypervisor Probes (hprobes). The hprobe framework allows us to monitor the execution of both the guest OS and applications from the hypervisor. To supplement this monitoring framework, we use dynamic analysis techniques to investigate the relationship between hardware events visible to the hypervisor and OS constructs common across OS versions. We use the results of this analysis to parametrize the hprobe-based monitors without requiring any user input. Combining the dynamic VM monitoring framework and analysis

frameworks allows us to provide on-demand hypervisor based monitors for cloud VMs.

To my family and friends, for their love and support.

Acknowledgments

Throughout my graduate career and life, I have been surrounded by wonderful people. If it weren't for the support I've received this dissertation would not exist - hence this thanks may be longer than is typical. I apologize to anyone that may have been omitted; be assured that any omissions are not due to a lack of appreciation. Any words I write do not give justice to the sense of gratitude that I feel.

Thanks go first and foremost to my mentor and advisor, Prof. Ravi Iyer. Your guidance and support have been essential to this work and my overall growth as a researcher and person. The wisdom to realize that just working on a problem will bring about new ideas got me out of the standstill in my research. I will continue to recommend that same advice to my students. You always provided sound guidance and kept my best interest in mind, even when I didn't like to hear it. Can we declare success?

This thesis would not exist without Prof. Zbigniew Kalbarczyk: despite all the head shaking I have caused, you showed extreme patience and have been an outstanding mentor. Your ability to ask the right questions of any presentation/paper/project is a skill I strive to develop. Thanks also for answering random language questions, and apologies that you now need another source of rye bread.

A piece of advice I received in graduate school was that you should get to know your committee, and I am glad that I did. Thanks to Prof. Michael Bailey for helping me put my work in context, Prof. Roy Campbell for his insight into how my work relates to computer science principles and other bodies of research, and Prof. Wen-Mei Hwu for his impressive ability to pick up on nuances in any work. I would also like to acknowledge Prof. Bill Sanders for his help and guidance over the years.

No one succeeds alone, and I have been supported by the best collaborators I could've hoped for. Thanks especially to Cuong for being a great colleague

and friend. Most of this work was the result of our collaboration, and I hope we can work together again soon. Special thanks to Dr. Lok Yan at AFRL for being a mentor, and also for introducing me to emulator-based dynamic analysis. I am extremely grateful for all your time, help, and guidance. Thanks to Read for all your help over the past year with bringing the RSaaS work to fruition: I look forward to our continued collaborations. Thanks to Phuong for sharing AttackTagger and also your input and work on the keylogger problem.

To my colleagues in the DEPEND group, I have enjoyed working with you over the years and I already miss our time together. I am especially grateful to Heidi for help in maintaining an environment of serenity and sanity. Thank you to Carol Bosley and our adopted member Carol Wisniewski, you were always willing to help and put up with my craziness. Thanks to Arjun for friendship, food, proofreading, and more recently for providing stays at hotel Athreya. Thanks to Mr. Stephens for sharing mad roasting and barista skills, not to mention explaining a thing or two about omics. Thanks to Key-whan for kindness and demonstrating the special patience that comes with sharing an office with me. Thanks to DEPEND group members past and present: Daniel, Eric, Fei, Gary, Homa, Hui, Keun-Yu, Lelio, Qingkun, Safa, Saurabh, Subho, Ted, Valentin, Valerio, and Yoga - I have enjoyed spending time with each and every one of you.

You may not know it from the previous paragraph, but CSL is more than just the DEPEND group and I am grateful for my interactions with many wonderful people in CSL. Thanks especially to Andrea, Amy, Adel, Atul, Ron, and Uttam.

I have had the pleasure to work with some talented undergraduate researchers. Thanks to Bhargava, Dennis, Lavin, Mika, Raj, Yihao, and Zhongzhi. While your projects did not always contribute directly to my thesis, working with you helped me understand my work and provided motivation in my own work.

I have made many wonderful friends in grad school. Particular mention goes to Tomek, a living example of determination and someone who can help with building a wooden box in a pinch, Brian for presentation help and sharing a contagious enthusiasm for education, and Michael for being generous and encouraging. White Horse Inn gets a shout-out for that awesome $1 well whiskey and half-priced burger special - you will be missed.

Special thanks to Maruja for everything you've done during these grad school years. I truly am a better person as a result of our relationship. For someone as stubborn as me you should consider that an accomplishment. This dissertation is in no small part due to your efforts. You have helped and sacrificed more than most people know. This is truly a triumph of courage, ingenuity, and teamwork. From the bottom of my heart, thank you.

Without my family, this dissertation would never have been completed - Not only since I would not be here, but because my family helped shape me into the person I am today. Thank you to my grandparents; the stories of your adversities and accomplishments remind me to appreciate the foundation for everything I have. Thank you to my brother Max for taking an interest in my research and for sharing my curiosity in how the world works. Thank you to my Dad for never letting me take a shortcut and for teaching me more than any class ever could. Thank you to my Mom for giving me everything I need and for believing in me even before I was born. Finally, thank God!

This dissertation is based upon work supported in part by the National Science Foundation under Grant No. CNS 10-18503 CISE and 13-37732 MRI, by the Army Research Office under Award No. W911NF-13-1-0086, by the National Security Agency under Award No. H98230-14-C-0141, by the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement No. FA8750-11-2-0084, by the Department of Energy under Award Number DE-OE0000097, by an IBM faculty award, and by Infosys Corporation.

Table of Contents

List of Abbreviations

ASLR	Address Space Layout Randomization
CFG	Control Flow Graph
CFI	Control Flow Integrity
CPL	Current Privilege Level
EPT	Extended Page Tables
EPTE	Extended Page Table Entry
EPTP	Extended Page Table Pointer
DOS	Denial of Service
GPA	Guest Physical Address
GVA	Guest Virtual Address
HAV	Hardware-Assisted Virtualization
HPA	Host Physical Address
IaaS	Infrastructure as a Service
IDS	Intrusion Detection System
iKGT	Intel Kernel Guard Technology
ISA	Instruction Set Architecture
KVM	Kernel-based Virtual Machine
MMU	Memory Management Unit
MSR	Model Specific Register
OS	Operating System

pid	Process IDentifier
PTE	Page Table Entry
QEMU	The Quick EMUlator
RSaaS	Reliability and Security as a Service
SMEP	Supervisor Mode Execution Prevention
SMM	System Management Mode
SMI	System Management Interrupt
TB	Translation Block (used in QEMU)
TCB	Trusted Computing Base
TDP	Two-Dimensional Paging
THP	Translation-Lookaside Buffer
TLB	Translation-Lookaside Buffer
vCPU	Virtual Central Processing Unit
VM	Virtual Machine
VMCS	Virtual Machine Control Structure
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor
VMX	Virtual Machine Extensions
VT-x	Vanderpool Technology (Intel Virtualization Technology)
WCET	Worst Case Execution Time

Chapter 1

Introduction

1.1 Motivation

Cloud computing allows users to obtain scalable computing resources to simplify setting up and running a scalable computing environment. However, cloud systems do not provide a mechanism by which users can protect their rented resources against accidental failures and malicious attacks. When it comes to making those systems reliable and secure, current cloud computing systems leave users in the same situation as if they were still running their own infrastructure. With growing threats and a rapidly changing landscape of attack and failure modes, what is needed is a method for alleviating the amount of effort and skill cloud users need in order to protect their rented resources.

Prolific failures have kept reliability as a foremost concern for customers considering the cloud [1]. Monitoring is especially important for security since many attacks go undetected for long periods of time. For example, Trustwave surveyed 574 locations that were victims of cyber attacks [2]. Of those 574 locations, 81% of the victims did not detect the attacks themselves (either a customer reported data misuse or a 3rd party audit uncovered a compromised system). In the cases where attacks were detected, the mean detection time was 86 days.

Cloud computing environments are often built on top of virtual machines (VMs) running on top of a hypervisor. A virtual machine is a complete computing system that runs on top of another system. The hypervisor is a privileged software component that manages the VMs. Typically, one can run multiple VMs on top of a single hypervisor, which is often how cloud providers distribute customers across multiple physical servers. As the low-level manager of VMs, the hypervisor has privileged access to those VMs and

this access is often supported by hardware-enforced isolation. The strong isolation between the hypervisor and VMs provides an opportunity for robust security monitoring. Because cloud environments are often built using hypervisor technology, VM monitoring can be used to protect cloud systems. However, existing VM monitoring systems are unsuitable for cloud environments as those monitoring systems require extensive user involvement to configure monitoring or when handling multiple operating system (OS) versions.

In this dissertation, we present a VM monitoring system that is suitable for cloud environments to provide Reliability and Security as a Service. By providing reliability and security monitoring as a service we can alleviate customers' concerns about their lack of awareness in cloud systems. In the system we envision, the cloud provider develops a set of monitors which are offered to the users. Since cloud providers have many experts on hand and vast experience with running large systems, providing as-a-service monitors in this fashion is a way by which that experience can be shared with customers without sacrificing the customers' independence.

1.2 Challenges

Failures and attacks demand a response beyond what current virtual machine (VM) monitoring solutions offer. Those monitoring systems often require guest operating system (OS) modifications, cannot be modified at runtime, and cannot monitor the execution of user programs. This lack of flexibility and high implementation complexity make many monitoring systems unsuitable for use in most IT environments.

There has been significant research on virtual machine monitoring [3–8]. Existing VM monitoring systems require setup and configuration as part of the guest OS boot process or modification of guest OS internals. In either case, the effect on the guest is the same: at the bare minimum a VM reboot is necessary to adapt the monitoring system; in the worst case the guest OS needs to be modified and recompiled. Operationally, these requirements are undesirable for a number of reasons, e.g. due to increased downtime or inflexibility when monitoring needs change.

In addition to the lack of runtime reconfigurability, VM monitoring is at a disadvantage compared to traditional in-OS monitoring in terms of infor-

mation available to the monitors. VM monitoring operates at the hypervisor level, and therefore has access only to low-level hardware information, such as registers and memory, with limited semantic information on what the low-level information represents (e.g., what function is being called when the `call 0xabcd` instruction is executed). In the literature, the hypervisor's lack of semantic information about the guest OS is referred to as the *semantic gap*.

1.3 Contributions

One class of active monitoring systems is a hook based system, where the monitor places hooks inside the target application or OS [5]. 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 transfered 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 as-a-service monitoring as hook-based systems are adaptable: 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 monitoring target (i.e., guest OS in the case of VM monitoring). Adaptability is particularly important in production IT environments, where monitoring needs to be configured for multiple applications and operational use cases. In addition to supporting a variety of environments, monitoring must also be responsive to changes in those environments (e.g., repurposing of computing resources).

In this dissertation, 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 supports runtime in-

sertion/removal of hooks. All of these aspects result in a VM monitoring solution that is suitable for as-a-service monitoring on an actual production system. We have built a prototype implementation using Hardware-Assisted Virtualization that is integrated with the KVM hypervisor (Section 2.4). To demonstrate monitoring using the hprobe framework, we constructed example dectectors: an emergency security vulnerability detector, a heartbeat detector, and an infinite loop detector (Chapter 5). While our prototype framework shares some similarities and builds on previous VM monitoring systems (Section 3.5), the example hprobe detectors could not have been implemented on any existing platform.

The hprobe framework is still subject to the semantic gap challenge introduced in the previous section. Namely, hprobe monitors still need a low-level definition of a hook location (i.e., an address in memory) as well as parameters for the monitoring functionality (e.g., timeout values for reliability detectors). In order to allow the hprobe system to be used in an as-a-service monitoring system, we developed a method for bypassing the semantic gap by abstracting across different versions of OSes through the use of OS constructs common across versions (Chapter 6). Those OS constructs are leveraged so that the hypervisor can access the appropriate information as necessitated by the reliability and security monitors. This information is then used to infer the VM's state: i.e., detect erroneous or abnormal behavior.

Monitors developed using our technique are based on higher-level OS concepts and are not dependent on version-specific data structure layouts or variable names. For example, while traditional VM monitoring techniques are effective for extracting a wealth of information from the guest OS, they are often dependent on not only the OS family, but also the OS version. If the data structure layout or structure member names change, the VM monitoring system will need to be updated. This means that a VM monitoring system for a cloud-based service would need to include every possible variation of data structure layout and type (e.g., structure members) for its customer VMs. Furthermore, for every set of offsets and types, there will also be a set of values that vary across more VMs (e.g., for each minor kernel version). In our approach the monitors still interact using low-level operations (e.g., instructions, registers, memory reads and writes) as that is the layer at which hypervisor-based systems operate. The detectors we design, however, do not need to understand the low-level information such as which

offset into a structure is the process ID; necessary low-level parameters are automatically inferred through the use of OS constructs.

A cloud provider can leverage its systems expertise to design monitors based on the frameworks developed throughout this dissertation and then offer those monitors to cloud users. VM monitors based on these techniques are suitable for a cloud environment. This alternative to traditional VM monitoring allows a cloud provider to support monitoring across OS versions, which is essential for a cloud environment where users have full control over their VMs. The key research contributions of this dissertation are:

1. A dynamic VM monitoring system based on hardware-assisted virtualization that is runtime adaptable,

2. A technique for obtaining low-level VM monitoring parameters that is based on OS constructs common across multiple versions of an OS,

3. A reliability and security monitoring framework that does not modify a cloud user's VM or require input from the user other than a VM image containing his or her specific OS and applications,

4. Sample monitors that demonstrate the range of monitoring one could deploy using the dynamic monitoring system and Reliability and Security as a Service framework.

Chapter 2

System Virtualization

2.1 Emulation and QEMU

In the goal of running an entire computing system on top of another, one of the most straightforward concepts is emulation: a piece of software that acts as a CPU reads the machine code from the image of a system and translates it to instructions that run natively on the host CPU. Emulation frequently sees use in contexts where one wishes to run software from one instruction set architecture (ISA) on another ISA (e.g., to run software compiled for a mobile phone that uses ARM on an x86 machine). However, in many cases one will simply wish to run a complete x86 system on top of another x86 system (e.g., for OS development). One open-source emulator that is very popular is the Quick EMUlator or QEMU [9]. The advantage of an emulator like QEMU over other virtualization techniques is that emulators do not require any hardware support and can also be used to run software from vastly different ISAs on one another (e.g., RISC on CISC).

QEMU uses the concept of a host ISA (the ISA for the CPU running QEMU on) and a target ISA (the ISA for the CPU QEMU is emulating). As of writing, QEMU supports 15 target ISAs.[1] A main design principle behind QEMU is portability; as such, QEMU does not have software to translate from every host to target ISA, but it uses an intermediary set of operations called TCG ops. TCG ops are a RISC-like ISA and are named after the software component, the tiny code generator, that performs the translation from the target ISA to TCG ops. At runtime, QEMU translates the target instructions into blocks of TCG ops called translated blocks (TBs). These TBs are related to basic blocks from compiler theory: a translation block is a set of instructions that end in a branch. Once a TB is translated from TCG

[1] http://wiki.qemu.org/Documentation/ISAManuals

```
x86 target input:
0x002f1400:  int    $0x80

TCG ops:
---- 0x2f1400
movi_i32 tmp4,$0x2f1400
st_i32 tmp4,env,$0x20
movi_i32 tmp12,$0x2
movi_i32 tmp13,$0x80
movi_i64 tmp14,$raise_interrupt
call tmp14,$0x0,$0,tmp13,tmp12

x86 host out:
0x425d3a00:  mov    $0x2f1400,%ebp
0x425d3a05:  mov    %ebp,0x20(%r14)
0x425d3a09:  mov    $0x80,%edi
0x425d3a0e:  mov    $0x2,%esi
0x425d3a13:  mov    $0x7f07384191a0,%r10
0x425d3a1d:  callq  *%r10
```

Figure 2.1: An example of translating from an x86 target to an x86 host using intermediary TCG ops in QEMU: the `int $0x80` instruction traditionally used to issue system calls on Linux. This code was generated using the `-d in_asm,op,out_asm` arguments when running QEMU.

ops into the host ISA, it is placed into a translation cache (default 32MiB in size). An example of the translation of one instruction (a legacy Linux system call using `int $0x80`) from target x86 into TCG ops and finally into host x86 is shown in Fig. 2.1.

From Fig. 2.1, we can see that one instruction in the target can lead to many instructions in the host, even when the architecture is the same as in the example. Note that the last TCG op and host instruction in the TB is a call to another function, `raise_interrupt`. The `raise_interrupt` function itself contains 123 instructions so one instruction in target x86 becomes 129 instructions in the host OS (note that this number was determined statically; the control flow of `raise_interrupt` may increase or decrease this amount). A significant contributor to this explosion in the number of instructions executed is handling interrupts and privileged operations: since QEMU is a user-level process it must translate all privileged operations (like interrupt handling and memory management operations) into user-level operations that

emulate and manage the hardware state (e.g., the `raise_interrupt` function).

The overall performance implications of emulation are significant: in addition to the inefficiencies of translating when a block does not exist in the cache and the increase in number of instructions one can expect cache performance to be impacted since the number of intermediary operations would affect locality.

2.2 Virtualization

In this dissertation, the term "virtualization" will always refer to the idea of system virtualization, or running a complete virtualized computing system (i.e., operating system) on top of a physical computer. Emulation allows one to run a complete virtual computer on top of a physical computer (even in the case of differing ISAs), but introduces many overheads as described in the previous section. When running a target and host of matching ISAs, it would be preferred to skip all the intermediary steps and run the target code directly on the host CPU (e.g., the `int $0x80` instruction should just execute `int $0x80`, not 129 other instructions). Virtualization is just that: running a complete virtual or "guest" computer system on the native CPU as much as possible.

The canonical work describing virtualization was published in 1974 by Gerald Popek and Robert Golberg in 1974 [10]. Despite being published in 1974, the Popek and Goldberg paper presented and discusses ideas that are relevant and developing today (e.g., what Popek and Goldberg called "Recursive Virtualization" is now called "Nested Virtualization," which was presented in OSDI 2010 [11] and added to upstream KVM in 2012).

To implement a virtualized environment, Popek and Goldberg presented the idea of a virtual machine monitor (VMM) that runs on top of the hardware and controls the underlying virtual machines (VMs). In his PhD thesis, Goldberg made the distinction between two classes of VMMs that are still in use today: the Type I hypervisor and the Type II hypervisor [12]. This model for a VMM and Type I/II VMMs are illustrated in Fig. 2.2. A Type I VMM is a VMM that runs directly on the physical machine, replacing the host OS. A Type II VMM is a VMM that runs on top of the host OS (in Goldberg's original work, he described a Type II VMM as running on an "ex-

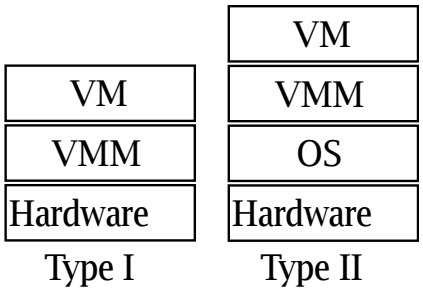


Figure 2.2: The simple model for a Type I and Type II Virtual Machine Monitor, as presented by Goldberg.

tended machine," an additional abstraction offered by the OS). In both cases, the VMM is responsible for managing virtual machines and plays a similar role for VMs as an OS does for user-level applications. In many cases, the VMM is referred to as the "hypervisor" since it has more control than the guest OS or "supervisor." The terms VMM and hypervisor will be used interchangeably throughout this dissertation, but hypervisor has become the more accepted term in enterprise IT and will be preferred in later chapters.

According to Popek and Goldberg, a virtualized system manage has the following characteristics:

1. *Essentially identical to bare metal*: the functionality of the VM must be the same as running on bare metal, modulo minor differences (e.g., the VMM can present a VGA card, but it need not offer a virtual version of all available hardware on the market).

2. *Efficiency*: A majority ("statistically dominant" in Popek and Goldberg's terminology) of instructions must be executed directly on the physical CPU. This is the main differentiator from emulation, which has the additional translation steps.

3. *Resource control*: The VMM must maintain control of the physical computing resources at all times: VMs have full hardware access while running, but the VMM is able to override VM operations (e.g., by preempting the VM's execution). This property has important implications that will be useful for reliability and security monitoring.

After presenting the above characteristics of virtualized systems, Popek and Goldberg proposed the "trap-and-emulate" method for implementing a VMM. In trap-and-emulate, the VM's instructions are executed natively

on the host CPU until a privileged operation is needed. Upon reaching a privileged operation, the execution of the VM traps to the VMM where it emulates the necessary functionality (e.g., updating page tables or performing disk output). Once the VMM is finished emulating the necessary functionality for a privileged operation, the VMM returns control to the VM which continues by natively executing instructions on the host CPU.

2.3 Virtualization and x86

While systems such as IBM's CP/CMS had hardware support for virtualization since the s/360 in the 1960s [13], Intel's x86 architecture did not have hardware support for virtualization until 2007. Before hardware support for virtualization, x86 was viewed as "unfriendly" to software based virtualization. For example, one cannot directly implement the trap-and-emulate method for x86 in software. This is due to the fact that not all privileged instructions executed with user privileges trap to the OS/kernel privilege level. For example, the `popf` instruction that populates the EFLAGS register with values from the top of the stack behaves differently depending on which privilege level the CPU is in:[2] when the CPU is executing in kernel mode, the CPU sets certain flags (e.g., IOPL), whereas when the system is executing in user mode, the CPU just leaves those flags unaffected. This behavior difference across privilege levels prevents a potential VMM from running a VM in user mode and just emulating any privileged operation when that privileged operation traps to the host OS (the host OS would effectively be functioning as a VMM in this case).

2.4 Hardware Assisted Virtualization in x86

Most of the techniques presented in this dissertation are built on Hardware-Assisted Virtualization (HAV) as offered in Intel x86. HAV is an alternative to other techniques such as paravirtualization (e.g., as used by Xen [14]) and binary translation (e.g., as used by VMware [15]). This section provides a

[2]For full details, see "Operation" at http://x86.renejeschke.de/html/file_module_x86_id_250.html

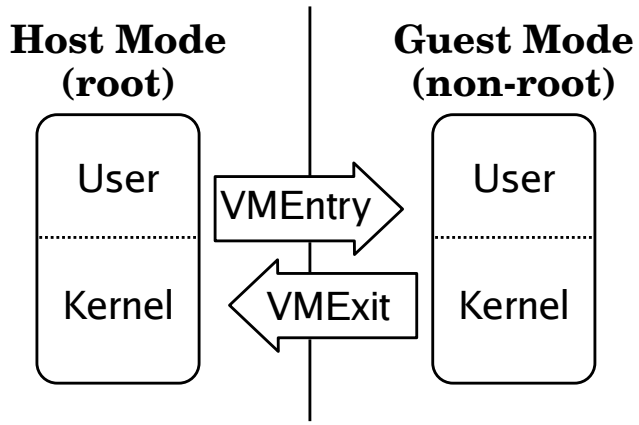


Figure 2.3: Intel VT-x adds another set of privileged modes multiplexed with x86 privilege levels: the VMM or hypervisor executes in Host Mode and the VM or guest executes in Guest Mode.

brief summary of Intel's Hardware-Assisted Virtualization (HAV) virtual machine extensions (VMX) to the x86 instruction set as needed to understand the work presented throughout this dissertation; complete details are available in the Intel Software Developer's Manual [16]. While the description of the hardware architecture below and work presented in this dissertation pertain to Intel VT-x, AMD's AMD-V technology is based on essentially the same concepts and details can be found in AMD's Architecture Programmer's Manual [17].

Intel's HAV implementation extends the x86 ISA with Virtual Machine eXtensions (VMX) that allow for one to develop a "trap-and-emulate" VMM/hypervisor as described in the previous section. All major operating systems executing on the x86 platform run in protected mode, and effectively use two privilege levels: ring 0 ("system mode") and ring 3 ("user mode"). Similarly, Intel VT-x uses two modes of operation: *root mode* and *non-root mode*, which are layered on top of rings 0 and 3. The hypervisor executes in root mode and the virtual machines (VMs) execute in non-root mode (often referred to as guest mode). The hypervisor runs a virtual machine (VM) by invoking a *VM Entry* (via the `vmlaunch` instruction). When the VM is executing in non-root mode and certain operations are performed (e.g., a privileged instruction is executed), control is transferred to the hypervisor (the CPU transitions to root mode) via a *VM Exit* event. The transitions between root and non-root mode via the VM Entry/Exit mechanism are illustrated in Fig. 2.3. The VM Exit mechanism allows a guest mode OS to

run in privilege ring 0 and is an implementation of the "trap-and-emulate" style of virtualization [10]. A hypercall is a request (similar to a system call in an OS) that can be issued by a VM so that the hypervisor can perform an operation on its behalf. In Intel VT-x, the `vmcall` instruction is used to issue hypercall. In addition to instructions and VM Exit events, VT-x introduced the Virtual Machine Control Structure (VMCS), a 4KiB structure that is used to manage each virtual CPU (vCPU) of a VM. The VMCS contains information such as the register guest and host register state and is used to both configure a guest's vCPU as well as store the host state to be restored upon a VM Exit (e.g., the host's register values).

In addition to virtualizing the CPU, a VMM/hypervisor also needs to virtualize the memory management subsystem. The x86 Memory Management Unit (MMU)'s main task is to provide virtual to physical address translation through the use of paging. The MMU automatically walks page table structures managed by the OS kernel. The kernel maintains page tables to give user processes a virtual address space. This virtual address space is defined by the set of page tables located at the current page directory base address (which is obtained from the `CR3` register). In a virtualized system there are four different types of addresses: the *guest virtual address* (GVA) - a virtual address provided by the guest operating system; the *guest physical address* (GPA) - the physical address of the virtual memory that the guest sees and is allocated by the host; the *host virtual address* (HVA) - a virtual address allocated by the host OS; and finally the *host physical address* (HPA) - the address of physical memory used by the hardware.

Just as with x86 CPUs, the x86 MMU is not virtualized: there is only one MMU for the system and it cannot be partitioned for multiple OSs. In order to maintain the resource control requirement for virtualized systems, one cannot give the guest OS control of the MMU since it cannot be shared. Therefore, in the first implementations of x86 HAV, guest page faults would cause VM Exits and the hypervisor would manage translations via shadow page table structures in the hypervisor. However, adding a VM Exit to the page fault handling path decreases memory performance. To reduce the number of VM Exits, two-dimensional paging (TDP) allows VMs to manage their own page tables (GVA to GPA translations) directly. Intel's TDP is called *Extended Page Tables* (EPT). With EPT, the hardware traverses an additional set of page tables managed by the hypervisor to translate from GPAs

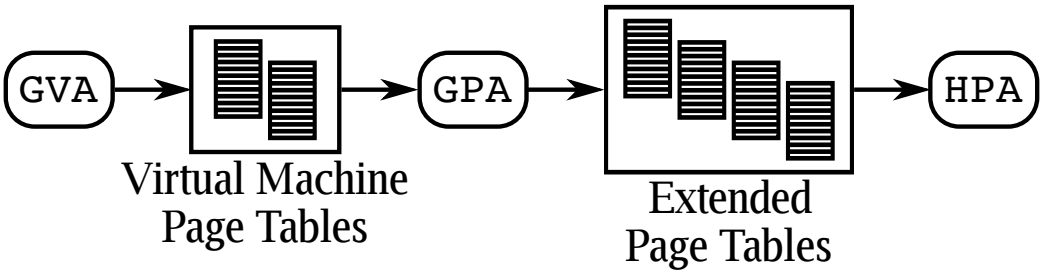


Figure 2.4: Translating a Guest Virtual Address (GVA) using Intel's Extended Page Tables (EPT). The Guest OS page tables (shown as a two-level structure) translate from guest virtual to guest physical, which is then translated to host physical by the EPTs (shown as a four-level structure).

to HPAs. Since the hypervisor generally operates with kernel privileges and can manage the physical address space, the HVA→GVA translation is not needed and would introduce unnecessary overhead. Since the guest OS can directly manage its own page tables and update `CR3`, EPT reduces the number of VM Exits due to page faults. However, in EPT, the hardware traverses two sets of page tables to translate a GVA: the guest page tables (stored in the VM's address space) are walked to translate from guest virtual to host physical address, and then the EPTs (stored in the hypervisor's address space) are walked to translate from guest physical to host physical address. This set of translations and the various paging structures are illustrated in Fig. 2.4. Note that while in EPT page faults are less expensive, Translation-Lookaside-Buffer (TLB) misses become much more expensive since additional page tables need to be traversed [18]. In the worst case with a 64-bit guest on a 64-bit host (64-bit page tables are 4 levels), a TLB miss with EPT will result in 20 memory accesses to translate a single GVA: 5 memory accesses are needed to translate each of the 4 levels in the guest page tables (4 accesses to EPT Entries to translate the guest page table entry + 1 access for reading the guest page table entry itself). Similar to Page Table Entries (PTEs), EPT Entries (EPTEs) describe a page and contain information like the address of the page and also control bit (e.g., for permissions). If an EPTE does not exist for a GPA or if the permissions for that guest physical page would be violated by a memory access, the CPU will VM Exit with an `EPT VIOLATION`, which is analogous to a page fault for a normal OS.

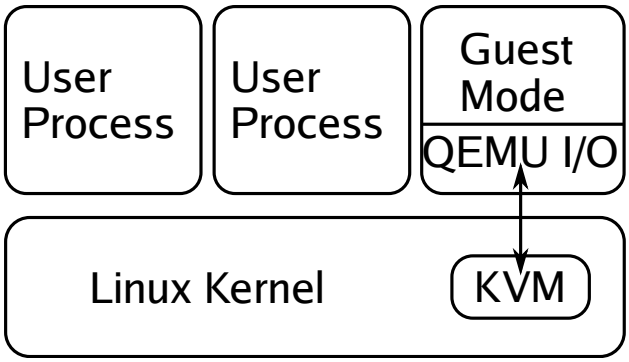


Figure 2.5: Typical KVM architecture: KVM itself is a module that can be loaded into the Linux kernel at runtime and a Virtual Machine is treated as a separate user process that usually utilizes QEMU for device emulation.

2.5 Kernel-based Virtual Machine

To meet community demand for a standard open-source hypervisor, the Linux community developed a hypervisor integrated directly in the Linux kernel. This hypervisor, called Kernel-based Virtual Machine or KVM [19], supports both Intel and AMD HAV. KVM is a standalone module for the Linux kernel and is often integrated with QEMU as shown in Fig. 2.5.

When using KVM, a VM is treated as a user process. In the most common use case, it is more accurate to refer to the hypervisor as *qemu-kvm* since the actual userspace program running inside the host OS is the same QEMU program that was described in Section 2.1. Both KVM and QEMU perform duties that are typically expected of a hypervisor, performing different types of emulation for trap-and-emulate virtualization. A VM is started with a QEMU process that performs all hardware emulation (e.g., sets up virtual hardware devices and runs a virtual BIOS or EFI). Once initial setup is complete, QEMU transfers control of the VM to KVM. Instead of translating instructions through the TCG as exemplified in Fig. 2.1, however, instructions are executed natively on the CPU through KVM's use of HAV. KVM follows the UNIX "everything is a file" philosophy and to use HAV instead of dynamic translation, QEMU communicates with KVM by issuing `ioctl` system calls on a special `/dev/kvm` device [20]. Each vCPU is treated as an open file and QEMU manages each vCPU through a file descriptor. After starting a vCPU with the `KVM_RUN` ioctl, KVM will issue a VM Entry and handle VM Exits pertaining to privileged OS-level emulation tasks (e.g., for

managing control register access and EPT violations). In some cases, such as for virtual devices such as an emulated disk drive, KVM cannot handle a VM Exit and KVM will forward those exits to QEMU by returning from the `KVM_RUN ioctl`. QEMU will then perform the necessary emulation task (e.g., write data to the disk) and reissue the `KVM_RUN` when it is ready to continue executing the VM using HAV. For a performance comparison of different hypervisor technologies including KVM, please see Appendix A.

Since KVM is a Linux kernel component and QEMU is an emulation program, both KVM and QEMU have a large number of configuration options that make it difficult to configure and manage VMs manually using those two components. Typically, system administrators use a management layer like libvirt to abstract away these details [21]. Libvirt provides an XML-based configuration interface that allows one to manage multiple hypervisors (e.g., Xen and KVM) using the same format and control interface.

2.6 Cloud Computing

An important use case of virtualization has been as the key building block of cloud computing systems. The most generally accepted definition for cloud computing was published by the US National Institute of Standards and Technology (NIST) in 2011 [22]. According to the NIST definition, the essential characteristics of a cloud service are:

- On-demand self-service: The user can automatically request capabilities from the provider.
- Broad network access: All systems must be accessible through the network.
- Resource pooling: Users benefit from large resource pools owned by the provider. The provider benefits by having maximum utilization of their systems.
- Rapid elasticity: Scale can be adjusted rapidly up and down to meet current demand.
- Measured service: Resources are provided in a utility model; both the user and the provider can monitor and control resource usage.

In addition to the above cloud characteristics, NIST also defined multiple cloud service models which have been widely accepted by industry. This dissertation considers the *Infrastructure as a Service* (IaaS) model (e.g., as offered by Amazon EC2). In practice, other cloud abstractions (e.g., Software as a Service/SaaS and Platform as a Service/PaaS) are often built upon IaaS. In IaaS, users rent computing resources from a cloud provider. The rented resources come in the form of compute, networking, and/or storage. At the base of these resources is compute, which is typically delivered in the form of Virtual Machines that the customer runs on top of the cloud provider's physical hardware. A key advantage of using IaaS as opposed to other cloud models is that the user has complete control over their environment. If an organization is migrating from their own data center to a cloud, IaaS offers the least disruptive migration path since other cloud models (e.g., PaaS) require one to redevelop an application for a new interface. When renting VMs from an IaaS cloud provider, users upload an image to the provider's platform or create it directly on that platform. If the cloud provider uses a standard format for VM images (e.g., the one preferred by open source tools such as QEMU), IaaS can also be viewed as a vendor-neutral solution.

Cloud computing environments can either be offered as public or private clouds. A public cloud is the typical cloud environment where one rents resources from a cloud service provider such as Amazon or Microsoft. A private cloud is one run by an organization for internal use, often with the intention of sharing development resources across multiple groups. A hybrid cloud is a combination of a public and a private cloud, where an organization uses a private cloud that connects to a public cloud should the local computing resources be exhausted.

Chapter 3

Hypervisor-based Reliability and Security Monitoring

3.1 The Hypervisor as Part of the Trusted Computing Base

The virtualization characteristics defined by Popek and Goldberg and presented in Section 2.2 allow one to develop robust reliability and security monitoring. In particular, the resource control requirement allows one to place a monitor at the level of the hypervisor/VMM that inspects the behavior of a potentially untrusted VM. Note that when performing hypervisor-based VM monitoring, the Trusted Computing Base (TCB) includes not only the hardware, but also the hypervisor. A high-level diagram explaining hypervisor-based monitoring is illustrated in Fig. 3.1.

Integrating the hypervisor into the TCB does mean, however, that VM monitoring is vulnerable to VM Escape and similar attacks [23]. A VM Escape attack is an attack where a VM breaks out of its isolated environment and affects either the hypervisor or a neighboring VM. However, in practice these attacks are rare (but notable, e.g., VENOM [24]), but addressing and

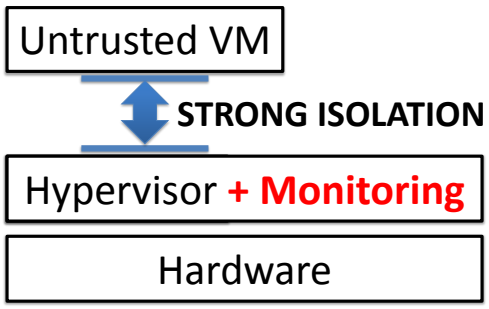


Figure 3.1: Illustration of Hypervisor-based Monitoring. One adds monitoring functionality to the hypervisor to take advantage of the strong isolation between the hypervisor and a potentially untrusted VM.

preventing them is an open research question. Most hypervisor vulnerabilities, however, do not occur in the actual HAV layer (e.g., KVM), but inside the device emulation components that are often implemented as privileged userspace applications (e.g., QEMU in the qemu-kvm configuration). For example, VENOM was a vulnerability in a virtual floppy device that was enabled by default in hypervisors that used QEMU for device emulation (e.g., KVM, Xen, and VirtualBox). One can mitigate against this risk by ensuring that only a minimal set of devices are loaded for a running VM as demonstrated by Nguyen et. al. in Delusional Boot [25]. However, this technique is not foolproof as other vulnerabilities may exist. Since a hypervisor is generally a small and well-defined codebase, it is amenable to a variety of integrity techniques [26–31].

In an IaaS cloud environment, one can expect a cloud provider to have a homogeneous data center, which an ideal environment for integrity verification techniques. It is not possible to make a similar claim for customer VMs based on homogeneity: clouds by their very nature have a highly varied workload across the entire customer base. This makes a technique like hypervisor-based monitoring particularly attractive: since the monitors are developed and sit at an abstraction layer below the customer's VM it is possible to reuse the same monitoring functionality across varying customer guest OSs and workloads.

3.2 Virtual Machine Introspection

The first hypervisor-based monitoring technique, virtual machine introspection (VMI), was introduced by Garfinkel and Rosenblum 2003 [3] and has been utilized for many purposes such as malware detection [32], building honeypots [33] and performing integrity checks on guest OS data structures [34]. VMI was presented as an intrusion detection method based on the VMware hypervisor. In their IDS work, Garfinkel and Rosenblum extracted OS properties using Linux's crash dump analysis techniques on the kernel memory image and then used this information to check for policy violations inside the guest OS. The main technique of VMI is to extract guest OS-level properties from the memory image of a running VM. Currently, the most popular VMI framework is the open source LibVMI framework [35]. However, to take low-

level information available to the hypervisor and perform actionable monitoring, VMI requires semantic information about the guest OS (e.g., kernel data structure addresses and offsets). The inability of the hypervisor to understand guest OS semantics without additional information about the guest is often referred to as the *semantic gap* in the literature.

In order to address the semantic gap, LibVMI creates a view of the guest OS by parsing the guest OS data structures by running code inside the guest OS (e.g., loading a kernel module that calculates kernel data structure addresses and offsets). After gathering those offsets, LibVMI obtains monitoring information of interest from the running VM (e.g., the list of current processes). In a homogeneous cloud environment, one would like to perform introspection automatically and without requiring one to compile and load special programs (e.g., kernel modules) inside the guest. To address the semantic gap in a more automated fashion, Virtuoso [36] and VMST [37] run standard UNIX utilities, such as `ps` or `lsmod` inside of a running VM. VMST runs the binary of interest on a clone of the guest in a trusted VM while redirecting kernel memory reads to the untrusted guest. All of libVMI, Virtuoso, and VMST require the provider to run code inside the user's VM (either a kernel module or application). A contribution of this dissertation is a means by which one can address the semantic gap without forcing the VM to run specialized code (see Chapter 6 for more details).

At the time of writing, LibVMI has been expanded beyond its traditional memory analysis to include support for hardware events and memory forensics [38]. LibVMI does support hook-based monitoring that is based on page permissions (e.g., using EPT with KVM on Intel).

3.3 Event-Based vs. Polling Monitoring

Broadly speaking, security monitoring can be decomposed into two steps: *logging* and *auditing* [39]. Logging is the act of recording data about a system and auditing is the processing of the logged data. A detector is a monitor where the logging and auditing phases are combined into one component: an action is observed and the decision is made immediately afterwards.

The logging phase of monitoring can be performed with two different schemes: polling and event-based. Polling-based monitoring is the tradi-

tional form of security monitoring: one periodically inspects the state of the system to log its activity (e.g., checking the list of running processes every 60 seconds). Event-based monitoring is similar to interrupt-driven I/O: an event is generated by a behavior in the system and logged. Event-based systems have an advantage over polling systems: in an event-based system any time an event occurs, the logger is notified because the logger subscribes to that event. Since the event is triggered upon the behavior of interest, it is possible to respond to the behavior immediately. Using this immediate response, by triggering on particular events we can potentially preempt attacks. This is contrasted with polling where an event may be detected after the fact.

In addition to concerns about responsiveness, polling-based systems present another issue: they are vulnerable to transient attacks. A transient attack is an attack that occurs between logging intervals in a polling-based monitoring system. To successfully execute a transient attack, an attacker must be able to launch an entire attack that is either shorter than the monitoring interval or can be decomposed into discrete stages (e.g., breaking apart a sensitive file and sending it over the network). Many polling monitors work by inspecting data structures inside a system (e.g., the list of running processes). The runtime of these monitors is therefore dependent on the amount of time it takes to inspect those data structures (both in the logging and auditing phase). If an attacker is aware of the monitoring systems in use, then that attacker can artificially increase the amount of work for the monitoring system to perform by spamming the system with fake data (e.g., spawning processes that perform no useful work). This spamming attack, along with a transient attack, is illustrated in Fig. 3.2.

When conducting a transient attack, it is important to understand the state of the monitoring system. That is, an attacker would like to know when logging is started so as to maximize his or her chance of success. In the case of an in-OS monitor, an attacker could determine that monitor runs by inspecting the list of processes and seeing when the monitor enters the runnable state and then calculating the monitoring interval [8, 40]. However, an advantage of hypervisor-based monitoring systems is the strong isolation of the monitoring system from the guest OS: it is not possible for a potential attacker to directly observe that a hypervisor-based monitor is running. Hypervisor-based monitoring systems that conduct polling can be especially vulnerable to transient attacks since the monitoring interval can be relatively

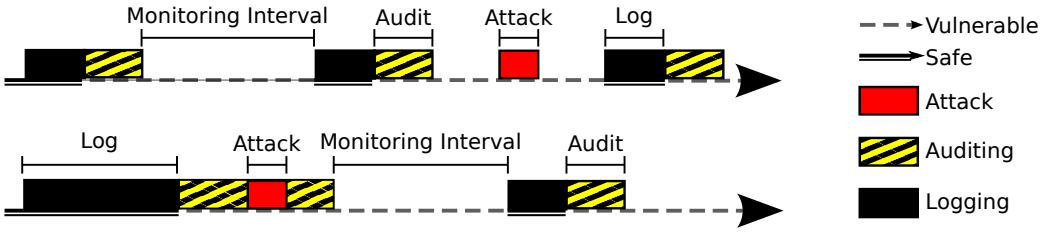


Figure 3.2: Illustration of transient and spamming attacks. A transient attack (top): the attacker launches an attack when a polling-based monitor is not logging and finishes the attack before the start of the next monitoring interval. A spamming attack (bottom): the attacker causes an attack to go undetected by creating extra work for both the logger and auditor, increasing the vulnerable window. Note that in this example, the attack actually occurs during the auditing phase, which is still vulnerable because the example monitor has serialized logging and auditing. © 2014 IEEE

long. This is especially the case if memory is inspected as with libVMI (e.g., 5 ms). This long inspection opens up hypervisor-based monitoring systems to timing-based side channels. In particular, we investigated the impact of the *VM suspend* timing side channel [41] that is caused when a hypervisor-based monitor paused a VM during monitoring. This side channel is illustrated in Fig. 3.3. Various techniques exist to mitigate the risk to a polling-based monitor from a timing side channel (e.g., virtual clocks [42] and randomized monitoring intervals), but these are most effective only against long attacks. Polling-based systems are fundamentally vulnerable to transient attacks and as such we chose to develop event-based loggers.

3.4 Hardware-based Techniques

In order to take advantage of virtualization's potential for monitoring, one can utilize events from HAV (i.e., VM Exits) to provide monitoring. Since VM Exits are guaranteed to occur for particular events by the hardware, they provide a robust mechanism for generating events that cannot be circumvented by actions inside the guest OS. Since VM Exits are tied to privileged operations that need to be emulated when the guest OS is performing certain operations, VM Exits offer insight into the behavior of the guest OS. Past work has demonstrated how hardware architectural state can be used to interpret a guest OS's operations [6, 43, 44]. In particular, Antfarm [43] and

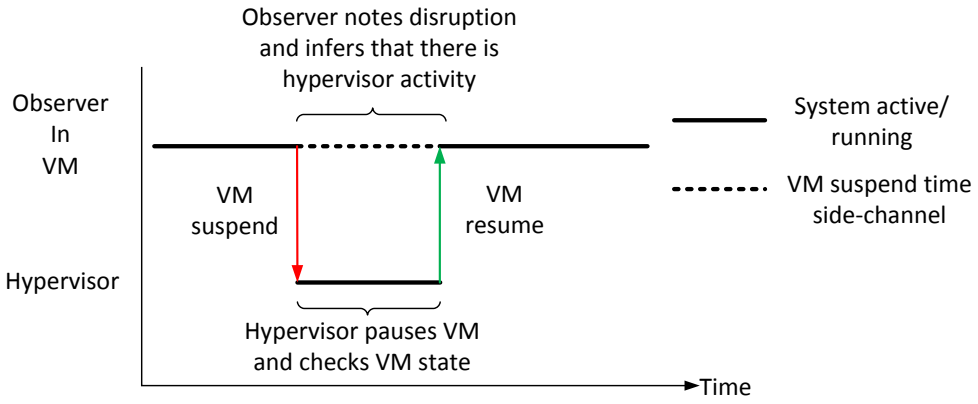


Figure 3.3: A diagram of how an attacker can measure the monitoring interval of a polling-based VM monitoring system in order to perform a transient attack. This figure was originally published by the USENIX Association *WOOT'15* [41].

its extension Lycosid [6] use `CR3` to count the number of guest user processes. Since `CR3` points to the active page tables for a process and each process must have its own virtual memory address space, each separate user process has its own `CR3` register. In pre-EPT configurations, Intel VT-x would generate a `CR_ACCESS` VM Exit event each time the guest OS overwrites the CR3 value (this VM Exit tells the hypervisor to change shadow page tables). The first work published with this technique was Antfarm, and the authors used VM Exit-based process tracking to build a hypervisor assisted anticipatory disk scheduler.

The same authors from Antfarm then extended the process counting idea to detect hidden process-level rootkits in Lycosid [6]. As part of a longer term attack, an attacker would often like to hide a process he or she has created (e.g., a keylogger). Many of these rootkits that hide processes will use techniques like Direct Kernel Object Manipulation (DKOM) that will manipulate the kernel data structures to hide the process from OS-level monitoring. Since VMI-based detection techniques involve traversing OS data structures from the hypervisor, these techniques would also fail to detect DKOM-based attacks (e.g., libVMI will also not see a process hidden from the task list). However, if the hidden process runs, it will still likely have its own set of page tables. Therefore, even if a process is removed from OS data structures

it will still cause a VM Exit when its `CR3` value is loaded. This VM Exit behavior allows us to construct a hypervisor-level view of the processes in a system. That hypervisor-level view can be compared with the OS-level view of the system and any inconsistencies can be indicators of a hidden rootkit.

It is important to note that the use of a unique set of page tables for each process is common across not only OSs that use x86 but other architectures. This OS behavior is invariant during correct operation of the system and is an example of a *hardware architectural invariant* since it is an OS invariant that is reflected in the hardware state. Our work in HyperTap defined the concept of hardware architectural invariants and provided examples of VM Exit-based detectors that have their root of trust in hardware architectural invariants [8]. In addition to developing (and testing on both Linux and Windows) a hidden rootkit detector similar to the one presented in Lycosid, we presented an OS hang detector based on `CR_ACCESS` VM Exits. OS hang detection through monitoring `CR3` writes is possible since during normal execution an OS will regularly be changing processes. Even if the system is in an idle state, one will still see system processes (e.g., init, watchdog, sshd) being scheduled. When one no longer sees `CR3` writes, processes are not being scheduled and the guest OS is hanging. Guest OS hang detection and hidden rootkit detection demonstrate the same hardware architectural invariant: that a process change is reflected by a `CR3` write can be used for both reliability and security monitoring. However, hardware architectural invariants have much broader use than just process counting and examples of other uses for reliability and security monitoring are tabulated in Table 3.1.

SecVisor is a small hypervisor built on Xen that protects the integrity of an underlying guest OS by ensuring not only kernel code integrity, but also by protecting core hardware data structures like the LDT and GDT [45]. Unlike a traditional hypervisor that is meant to manage multiple guest OSs, SecVisor is intended to protect a single guest OS using AMD's AMD-V virtualization technology. Since it is based on the AMD architecture, SecVisor uses the functionally equivalent Nested Page Tables (NPT) instead of Intel EPT to create separate isolated address spaces for kernel and userspace execution. In these address spaces, certain actions across address spaces are restricted (e.g., user code cannot be executed when executing in kernel space and kernel code cannot be executed when executing in user space). In order to learn about guest OS semantics, SecVisor requires a user to modify the

Table 3.1: Hardware Architectural Invariants from HyperTap © 2014 IEEE

Monitoring Category	**Guest event**	**Related VM Exit**	**Architectural Invariant**
Context switch	Process context switch	CR_ACCESS	The CR3 register always points to the PDBA of the running process Writes to CR registers cause CR_ACCESS VM Exits
	Thread switch	EPT_VIOLATION	The TR register always points to TSS structure of the running process TSS.RSP0 is unique for each thread
System call	Interrupt-based system call	EXCEPTION	Software interrupts cause EXCEPTION VM Exits
	Fast system call	WRMSR EPT_VIOLATION	SYSENTER's target instruction is stored in an MSR register Write to MSR registers causes WRMSR VM Exit
I/O Access	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 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	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

guest kernel code by adding hypercalls that inform the hypervisor of certain guest OS operations. Intel has recently open-sourced a project similar to SecVisor called kernel guard (abbreviated iKGT for "Intel Kernel Guard Technology") [46]. Like SecVisor, iKGT is intended to protect a single underlying guest OS. iKGT is intended as a policy enforcement framework and example policies also have features like process counting used by Lycosid and HyperTap.

As an alternative to using VT-x features for creating an isolated monitoring environment, SPECTRE uses Intel's System Management Mode (SMM) to protect the OS [47]. SMM is a special privileged mode of Intel CPUs that executes independently of the OS. SMM is often invoked using System Management Interrupts (SMIs), which are used by hardware vendors to perform OS independent functions (such as advanced power management functionality and memory prefailure monitoring [48]). SMIs can be invoked by the hardware and SMI handlers are completely isolated from the OS. SPECTRE is implemented as a set of SMI handlers that perform integrity checks on OS data structures. However, since SMIs are completely decoupled from OS execution and are executed periodically by the hardware, SMM-based techniques are vulnerable to transient attacks. Furthermore, SMM is difficult to work with and is not easily adapted: code used in SMM must be loaded by the BIOS at boot time and can conflict with existing vendor firmware (most enterprise IT environments would not want to void vendor warranties by overwriting BIOS). In the case of SPECTRE, the authors loaded a custom BIOS based on Coreboot [49]. Transitions into SMM are expensive and avoiding this performance cost limits what can be done [50]. Similar to detecting polling-based VMI monitoring, work has been performed by the low-latency community to detect unwanted vendor SMIs [51].

3.5 Hook-based Techniques

In order to provide as-a-service functionality, we require a monitoring system that can be enabled/disabled without disrupting the VM (to adapt the monitoring as user demands change) and that allows for monitoring of arbitrary operations (to allow any type of functionality to be developed on top of the monitoring system). Hook-based monitoring satisfies those requirements.

Hook-based monitoring works by replacing a part of the monitoring target's (e.g., a VM) execution with a hook (e.g., by overwriting a statement with a function call). When the hook is executed, control is transferred to the monitoring system. In a hypervisor-based hook monitoring system, the VM transfers control to the hypervisor when a hook inside the VM is executed. This is similar in operation to a debugger that briefly pauses a program to inspect its state. The research community has produced a variety of techniques for hook-based VM monitoring [5, 7, 52, 53].

An illustration of a hook-based monitoring system adapted from the formal model presented in Lares [5] is shown in Fig. 3.4. In the case of hypervisor-based VM monitoring, the target is a virtual machine and the monitor can run in either the hypervisor [4, 53], in a separate security VM [5], or in the same VM [7]. Regardless of the monitor's location, one must ensure that the monitor is resilient to tampering by the target VM in order to maintain the resource control requirement of virtualization. The monitor also must have access to all relevant state of the target VM needed to perform its monitoring function (e.g., virtual device information, guest memory). In order to support a wide range of monitoring, a VM monitoring system should be able to hook the execution of any instruction, be it in the guest OS or in an application. When looking at previous hook-based VM monitoring work, of particular note are the Lares [5] and SIM [7] approaches. Lares uses a memory-protected trampoline inserted by a driver in the guest VM. That trampoline issues a hypercall to notify a separate security VM that an event of interest has occurred. This approach requires modification to the guest OS (albeit in a trusted manner), so runtime adding and removing of hooks is not possible. Furthermore, a guest OS driver and trampoline is needed for every OS and version of OS supported by the monitoring infrastructure. The Secure In-VM Monitoring (SIM) approach uses a clever configuration of HAV by introducing entry and exit gates into a separate address space within the target VM. This in-VM approach prevents VM Exits for monitoring events and switches to a protected page inside the VM that performs monitoring. Since SIM does not incur VM Exits, it achieves low overhead. However, the additions of special entry and exit gates to the guest OS modify OS functionality at compile time. Hooks are also placed in specific predefined kernel locations, reducing adaptability.

In contrast to both the Lares and SIM approaches, hprobes (which are the

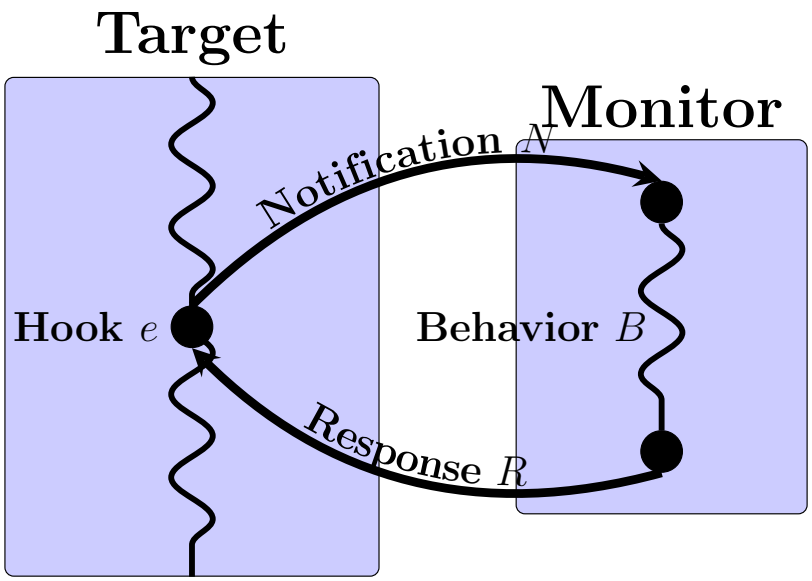


Figure 3.4: Hook-based monitoring. A hook triggers based on event e and control is transferred to the monitor through notification N. The monitor processes e with a behavior B and returns control to the target with a response R. In the case of hypervisor-based VM monitoring, the target is a VM and the monitor runs in the hypervisor. © 2015 IEEE

subject of Chapter 4) do not require any modification of the guest OS and can be added at runtime to arbitrary locations inside guests [53]. Note that no past work on hook-based VM monitoring mentions the removal of hooks, functionality that is necessary for an as-a-service approach and supported by the hprobe framework. While most research has been performed on platforms built on top of open-source technology, similar dynamic monitoring solutions also exist in proprietary systems. For example, VMware ESXi [52] has been monitored to provide similar functionality in vprobes.

3.6 Emulator-based Dynamic Analysis

3.6.1 Program Analysis

Broadly speaking, dynamic program analysis, or dynamic analysis, is the technique of analyzing a program by executing it [54]. Results from dynamic analysis are a true representation of how the program executes since one

executes the program during analysis. For example, the popular tool Valgrind [55] is a dynamic analysis tool used for debugging memory issues and also application profiling (e.g., for understanding performance characteristics such as cache behavior).

In contrast to dynamic analysis, static analysis is the technique of analyzing a program without executing it. Static analysis includes compiler-based techniques and source code analysis. Static analysis typically considers all possible control flow paths in a program and is commonly used for finding bugs in source code. Tools like the LLVM frontend Clang [56] provide this type of functionality.

Compared to dynamic analysis, static analysis techniques are typically not sensitive to any particular input, but can take longer to complete since one must examine multiple possible control flow paths. That said, while dynamic analysis provides precise information about execution of a program, it can be difficult to determine how general the results are [57].

3.6.2 Emulator-based Dynamic Analysis

An important use case for dynamic analysis tools has been malware analysis. Malware represents some of the most complicated software as malware often involves interactions with various aspects of a system (e.g., libraries and OS components). Furthermore, core malware functionality often relies on the behavior of dynamic memory management (e.g., in overflow-based attacks). Particularly sophisticated malware will use self-modifying code, which can be difficult for static analysis techniques to identify. While static analysis is valuable for verifying program correctness and finding bugs, it is not as well suited to programs that have complex system interactions, dynamic memory effects, and code that changes at runtime.

As discussed in Section 2.1, whole-system emulation represents an entire computing system in a software application. This software representation of a system is an ideal platform for measuring the complex relationships between hardware and software components. As such, emulator-based dynamic analysis tools are widely used for malware analysis [58–60]. While malware analysis is an important motivating use-case for emulator-based dynamic analysis, these emulator-based tools are ideal for analyzing other software

that has system-wide interactions and dynamic runtime properties. One class of software that also shares those properties is operating systems. As will be discussed in Chapter 6, we will use emulator-based dynamic analysis to supplement our VM monitoring system.

Emulator-based dynamic analysis tools are powerful and flexible, but have two major drawbacks: performance and detectability. While QEMU is a fast emulator, it still runs at least a factor of 4 slower than bare metal [9]. As researchers develop more tools to analyze and defeat malware, malware authors also devise new techniques to avoid detection. Since an emulator typically offers specific virtual devices (e.g., BIOS) and also has measurable performance characteristics, it is fairly straightforward for an application to detect when it is running inside an emulator. As many analysis tools are written using emulators, sophisticated malware changes its behavior when it detects it is running in an emulated environment. To address the performance and detectability drawbacks of emulator-based dynamic analysis, researchers have used features from leading CPU vendors to enhance dynamic analysis. Since emulators are popular for dynamic analysis, it is natural to extend those concepts to the VM Exit mechanism provided by HAV. VMMs specifically intended to be used as dynamic analysis platforms have been developed [44,61]. However, HAV is limited to a small subset of events in a system and if one wishes to have the same functionality as an emulator-based tool (e.g., to perform instruction-level analysis), any performance benefits disappear due to the large amount of context switching between the hypervisor and VM. Furthermore, virtualized systems suffer from many of the same detectability issues as emulators, though their improved performance and ability to directly use physical devices [62] make virtualized systems less detectable than emulated ones (e.g., KVM can present a vCPU with the same make and model of the physical CPU, whereas QEMU will present a very basic x86 CPU). In order to avoid detection by malware from running in either an emulator or VMM, branch tracing features have been used to produce traces of malware running on bare metal systems [63]. Branch tracing allows one to develop an analysis tool that runs on information that is collected directly from running an application on a physical CPU. However, these CPU features offer limited functionality when compared to emulator-based techniques and it is unclear that malware would not also be able to detect that the branch tracing features are enabled.

3.6.3 Dynamic Executable Code Analysis Framework (DECAF)

In this dissertation, we use the Dynamic Executable Code Analysis Framework, or DECAF, developed by the SycureLab at Syracuse University [60].[1] Like other emulator-based dynamic analysis frameworks, DECAF is based on QEMU and therefore it supports a VM image format that is compatible with the popular Xen and KVM hypervisors. DECAF works using a simple plugin API, has performance as a design principle, and has been demonstrated to work with both Linux and Windows guests.

DECAF is a modified version of QEMU and starting a VM in DECAF is just the same as using vanilla QEMU. In order to perform dynamic analysis in DECAF, one builds a plugin that registers callbacks on certain events. Plugins can be attached or removed from DECAF at runtime, allowing one to change analysis functionality without needing to restart the emulated system. Furthermore, certain callbacks can introduce significant overhead. To demonstrate the broad functionality offered by DECAF, a list of available callbacks at the time of writing is in Table 3.2. A key feature of DECAF is that it injects its callbacks into the Translation Blocks (TBs) generated by QEMU: i.e., callbacks are injected as function call TCG ops (refer to Section 2.1 for details of QEMU implementation). Since DECAF events are added to TBs, DECAF can be used as a cross-platform dynamic analysis tool. Though our work focuses exclusively on x86, DECAF has been extended for analysis of the Android OS running on ARM [64].

Note that while DECAF callbacks are invoked by function calls during TB execution, the callbacks actually pertain to architecture specific instructions and not TCG ops (e.g., the `DECAF_INSN_END_CB` callback would only be invoked once for `int $0x80`, not for every TCG op representing `int $0x80` in Fig. 2.1).

[1] At the time of writing, DECAF is available at: https://github.com/sycurelab/DECAF

Table 3.2: DECAF Callbacks

Callback Name	**Event on which it is invoked**
DECAF_BLOCK_BEGIN_CB	Called before a Translation Block (TB) executes
DECAF_BLOCK_END_CB	Called after a TB executes
DECAF_INSN_BEGIN_CB	Called before an instruction executes
DECAF_INSN_END_CB	Called before an instruction executes
DECAF_MEM_READ_CB	Called when memory in an address range is read
DECAF_MEM_WRITE_CB	Called when memory in an address range is written
DECAF_KEYSTROKE_CB	Called when a keypress event is sent to the virtual keyboard
DECAF_NIC_REC_CB	Called when the virtual network interface card receives a packet
DECAF_NIC_SEND_CB	Called when the virtual network interface card sends a packet
DECAF_OPCODE_RANGE_CB	Called when an instruction with an opcode falling within a specified range is executed
DECAF_TLB_EXEC_CB	Called when a TLB entry is added
DECAF_READ_TAINTMEM_CB	Called when tainted memory is read
DECAF_WRITE_TAINTMEM_CB	Called when tainted memory is written
DECAF_VMCALL_CB†	Called when the `vmcall` hypercall instruction is invoked
DECAF_INTR_CB†	Called when a hardware interrupt occurs
	†: callback was added a part of this dissertation

3.7 Evaluating VM Monitoring with User State Modeling Systems

While many of the detectors that will be presented in later chapters focus on deterministic monitoring (e.g., detecting an event that uniquely identifies and attack or failure), VM monitoring could also be used to provide information for IDSes [65, 66]. In this section, we seek to understand how useful VM monitoring data is as a source of monitoring information on a real system. In order to evaluate the potential effectiveness of VM monitoring in an IDS setting, we use historical data and a user state modeling intrusion detection system.

User state modeling systems are intrusion detection systems (IDSes) that work by tracking a user's actions inside a computing system. The system classifies users based on their previous actions and updates its classification as events occur in the system. For example, one can maintain a set of discrete states for each user in the system and classify those users as one of Benign, Suspicious, or Malicious. For our evaluation, we use the AttackTagger IDS which implements a user state modeling system based on factor graphs [67].

AttackTagger builds a factor graph for each user as events come in from various sources (e.g., network monitoring, host syslogs). The factor functions are predefined to describe the relationships between the new event, previous events, the user state, and the previous user state. Inference is then performed on the factor graph (e.g., using Gibbs sampling) to determine the most probable user state after each event. If the most probable user state at any time is Malicious, then AttackTagger raises an alert.

3.7.1 Simulations on Historical NCSA Incident Data

In the original AttackTagger paper, anonymized data from 116 security incidents occurring from 2008-2013 at the National Center for Supercomputing Applications (NCSA) was used to evaluate the effectiveness of its inference. The events were both manually and automatically generated from a variety of sources including syslogs, network monitoring logs, and the incident reports themselves. To test the effectiveness of VM monitoring as a source of information for an IDS, we reran those experiments using subsets of events that would be available from hypervisor monitoring (e.g., host based OS

Table 3.3: AttackTagger Evaluation of Monitoring Sources

Alerts Used	Hprobe	Network	Hprobe ∪ Network	All
TPR	52.63	65.52	74.58	79.03
TNR	99.25	91.30	98.20	98.48
FPR	0.75	8.70	1.80	1.52
FNR	47.37	34.48	25.42	20.97

events that could be detected using our monitoring systems). To compare with the existing tools currently in place, we also ran experiments with a subset of events that could be detected using network-based IDS tools [68]. For all experiments, AttackTagger was trained on the 51 incidents observed from 2008-2009 and tested on the 65 incidents observed from 2010-1013. The True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR), and False Negative Rate (FNR) from our experiences based on different subsets of events are shown in Table 3.3.

The "All" event set comprises all 83 events used in the AttackTagger experiments.[2] The "Hprobe" event set in Table 3.3 contains 28 alerts that could be generated using the hprobe VM monitoring framework (which is described in Chapter 4). The "Network" event set shows the result of the same data from only including the 37 events available to a network IDS, as this is one of the more widely deployed tools by NCSA. The "Hprobe ∪ Network" is AttackTagger running with union of both the "Hprobe" and "Network" event sets, totalling 60 events (there are 5 events that overlap in the two sets - e.g., opening an ssh connection by running the "ssh" command locally). We see that the information from VM monitoring does reduce the number of detections, but still maintains a low false positive rate. While the "HProbe" event set represents roughly one-third of the possible events, it is still able to detect roughly half of the attacks. We attribute this to the fact that VM monitoring is host-based and therefore VM monitoring detects actions directly occurring on a host with high fidelity. However, many types of attacks cannot be detected from using only host-based data and we see that the network-based IDS provides additional information to support VM monitoring. We see that the majority of attacks that can be detected using AttackTagger using both the VM and network-based monitoring alerts, and

[2]Note that the detection rates differ slightly from those published in [67] as the event sets in the codebase have changed since the publication of the original AttackTagger paper

those alerts represent 60/83=72% of the alerts. We note that when subsetting events, we took a pessimistic approach to get a lower bound for VM monitoring's detection quality. Specifically, unless we have sufficient information to suggest that an event is detectable by VM monitoring, we assume it cannot be detected. For example, we excluded events like FTP traffic from the "Hprobe" data set, whereas these events could be detected by VM Monitoring in certain cases (e.g., if a command line utility was used to initiate ftp communication, a monitor watching the process creation system call could potentially detect that FTP connection).

All of the experiments to generate Table 3.3 use historical data collected using the monitoring technology available at the time and also human descriptions in incident reports. For many incidents, a VM monitoring system may have been able to provide more information than what was recorded. If a VM monitoring system was deployed, it likely would perform better than what is presented in these experiments. We also note, from reading incident reports, that there is a significant amount of manual inspection that occurs in practice: when certain alerts are received, a security engineer logs into a machine and runs various commands to gather data or add additional monitoring software. One intention of developing an as-a-service monitoring system is to automate this inspection and addition of monitoring. For example, as a user transitions states from Benign to Suspicious on a system, one could increase the number of monitors for that system. Our dynamic monitoring system presented in Chapter 4 supports adding and removing monitors at runtime without disrupting the system.

Chapter 4

Hypervisor Probes (HProbes)

4.1 Dynamic Hypervisor-Based Monitoring

Two key components of an as-a-service system are on-demand self-service and rapid elasticity (see Section 2.6 and [22]). In order to support those key aspects of RSaaS, a monitoring service must be able to change at runtime. Runtime adaptability is particularly important for security detectors, where the landscape of possible attacks and vulnerabilities can change unpredictably. As new vulnerabilities and bugs surface, one will inevitably need to account for them in their monitoring infrastructure. In this dissertation, a *dynamic* monitoring system is a monitoring system that is adaptable at runtime. Dynamic monitoring not only allows one to enable and disable monitoring at runtime, but also to change monitoring functionality. As introduced in the previous chapter, we use a hook-based monitoring system to accomplish dynamic monitoring.

Contrasted with a dynamic monitoring solution, a static monitoring system is one which has its monitoring functionality hard-coded. In static monitoring systems, a compile time change is required to reconfigure its monitoring. In the case of static hook-based monitoring, this reconfiguration can either be hook locations, monitoring functionality, or both. When it is deployed, a static monitoring system is normally customized for its environment. However, the value of a static monitoring system decreases drastically over time unless periodic software updates are issued. Many VM monitoring solutions use a static monitoring approach [3, 5, 7, 8], and updates to monitoring functionality require code changes to either the hypervisor or guest OS. Those code changes would require a hypervisor reboot or a guest OS reboot in the best case. Reconfiguration 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 a performance improvement over statically configured monitoring: one can often monitor only an event of interest vs. a general class of events. This is usually because static monitoring systems may overcome inflexibility by over-subscribing to events of interest. For example, in order to monitor the underlying guest OS, one may use a static monitoring system with a hook into the guest OS system call handler (e.g., the `system_call` function) and use post-processing to filter out data on specific system calls of interest (e.g., the `exec` system call used to start processes). In a dynamic monitoring solution, one could just add hooks to the system calls of interest as needed (e.g., only hook the `sys_exec` function that handle the `exec` system call). Furthermore, it is possible to construct dynamic detectors that change during execution (e.g., a handler for one hook can be used to add or remove other hooks). This idea will be explored in the context of user state modeling systems in Section 3.7

Static monitoring systems embody a subtle violation of design principles: a monitoring target should not have its functionality tightly coupled to the monitoring system. However, a configuration change in a static monitoring system can disrupt the control flow of the target system (e.g., by requiring a restart of the underlying hypervisor). One can use techniques such as live migration to mitigate these shortcomings in a static system. However, live migration may not be suitable in large scale environments that need to adapt monitoring functionality quickly (e.g., to address a newly announced vulnerability).

In line with adaptability and being loosely coupled to the target system, a monitoring system should also be simple in its implementation. A security solution is only valuable if it is usable. This simplicity requirement extends not only to the implementation itself but also to the interface used to adapt monitoring (i.e., how one connects and develops new detectors). 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

[1] https://tools.ietf.org/html/rfc2535

[2] https://www.nsa.gov/public_info/press_room/2001/se-linux.shtml

(i.e., authentication and access control), both of those systems were released around the year 2000 and to this day are still disabled in many environments (e.g., instructions for installing software on Red Hat Enterprise Linux often start with "ensure SELinux is disabled"). In addition to increased usability and flexibility, a simpler implementation should yield a smaller codebase and therefore a smaller attack surface [69].

While decoupling a monitoring system from its target could impact the adoption of such a system from an ease-of-use standpoint, there are other implications for cloud environments. A key advantage of using a VM-based IaaS cloud over another hosting service (e.g., a virtual private server or container-based IaaS) is the ability to run a particular guest OS and have full control over that OS. From a cloud provider's perspective, that means that only a few assumptions can be made about the underlying guest OS (i.e., that the guest OS supports the host ISA). If a cloud provider wants to offer RSaaS, that monitoring system cannot be tightly coupled to the guest OS. Using a monitoring system that requires guest OS modifications raises maintenance efforts due to the number of guest OSs the provider must support. In addition, these monitoring systems also prevent cloud providers from monitoring VMs on-demand (support for guest OS must be enabled before monitoring can be used). In the worst case, a cloud provider can require the users themselves to modify their guest OS disk image. Requiring guest OS modifications also reduces one of the main advantages of using a HAV hypervisor: the ability to run an unmodified guest OS inside a virtualized environment. Furthermore, it has been shown that a large number of issues in cloud-based systems occur during system startup [70]. Requiring users to reboot the system upon every monitoring configuration change introduces increased risk with every guest OS or hypervisor reboot required to reconfigure those monitoring systems.

4.1.1 Design Principles

In order to satisfy the needs of a hypervisor-based monitoring system, we set the following design principles for a dynamic VM monitoring system:

1. **Protection:** Monitoring should be impervious to attacks (e.g., hook circumvention) originating from inside the VM, assuming that an attacker can own the guest OS. The authors of Lares [5] outline a formal

model with potential attacks and security requirements for a hook-based monitoring system. Those requirements using the notation in Fig. 3.4 from the previous chapter 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. While difficult to quantify, this requirement essentially aims to coincide with a philosophy of usable security. 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 at most loosely coupled to the target: reconfiguration can be expected to affect execution time, but it should not disrupt the control flow of the target (e.g., require a hypervisor reboot or guest OS restart). Furthermore, it should be possible to insert the hooks into both the target OS and its applications. Disabling monitors should be equivalent to removing the monitors, not merely ceasing to record logged events.

4. **Performance:** The monitoring system should have acceptable overhead for use in a production system. What is considered acceptable overhead is always application dependent.

We use these requirements as a guide to design a hook-based hypervisor monitoring framework that we call hypervisor probes or *hprobes*. As discussed in Chapter 3, the hypervisor provides a convenient interface for isolating monitoring from attacks and failures inside a VM while maintaining full access to the target VM's guest OS and applications. The hprobe 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 without disrupting the VM's control flow. In the remainder of this chapter, we discuss the implementation of the prototype and three small example monitors. Two of the monitors implement classical fault detection techniques and the third illustrates the use of dynamic monitoring to rapidly produce a detector that protects against a security vulnerability. More "realistic" hprobe examples will be discussed in the later chapters.

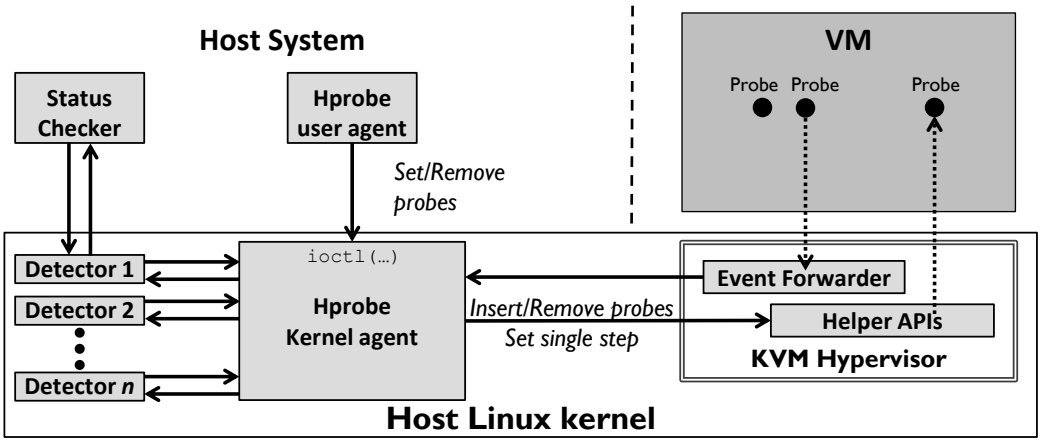


Figure 4.1: The hprobe framework integrated with the KVM hypervisor. The Event Forwarder passes valid hprobe debug events to the Hprobe kernel agent has been added by making code changes to KVM. The Hprobe kernel agent communicates with KVM through Helper APIs to add/remove probes. Detectors can either be implemented as kernel modules in the Host OS or in userspace by communicating with the kernel agent through `ioctl` functions. © 2015 IEEE

4.2 Hprobe Implementation

4.2.1 Integration with KVM

The hprobe framework was inspired by the Linux kernel profiling feature kprobes [71], which has been used for real-time system analysis [72] and serves as the foundation for the Linux troubleshooting toolkit systemap [73]. The operating principle behind the hprobe 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 [19]. The prototype's architecture is shown in Fig. 4.1. The modifications to KVM itself make up the Event Forwarder, which is a set of callbacks inserted into one of KVM's VM Exit handlers.[3] 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, containing the logic for probe execution and enabling/disabling of probes.

[3]Specifically, the `handle_exception` function that processes CPU exceptions such as debugging events

The kernel agent provides an interface to detectors (typically developed as kernel modules) for adding and removing probes. This interface is accessible by kernel modules through a kernel API in the host OS (as explained in Fig. 2.5 the host OS in this case also functions as a hypervisor since KVM itself is a kernel module) or by user programs via an `ioctl` interface.

The handling of hprobe execution by a detector is illustrated in Figs. 4.2 and 4.3. 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. We reconfigure KVM so that the `int3` instruction generates an exception when executed. Once `int3` generates a VM Exit event, 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 the exception was generated by an hprobe, 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 forwarded to the guest OS and handled as usual. Each hprobe handler performs a user-defined monitoring function and runs in the Host OS. When the handler returns (if desired, a deferred work mechanism can also be used to support non-blocking probes), the hypervisor replaces the `int3` instruction with the original opcode and places the CPU in single-step mode. Once the original instruction executes, a single-step (`#DB`) exception is generated, causing another VM Exit event (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 no probe events are missed (i.e., the hypervisor control the VM during probe handling). If one wishes to protect hprobes from being overwritten by the guest, the page containing the probe can be write-protected using TDP (e.g., EPT or NPT). Similarly, if one wanted to hide the `int3` opcode from casual inspection, a similar procedure can be used by read protecting the page and replacing the int3 with the original instruction (note that this would still leave a timing side channel open that may detect the presence of the probe). Although this prototype was implemented using KVM, the basic concept extends to any hypervisor

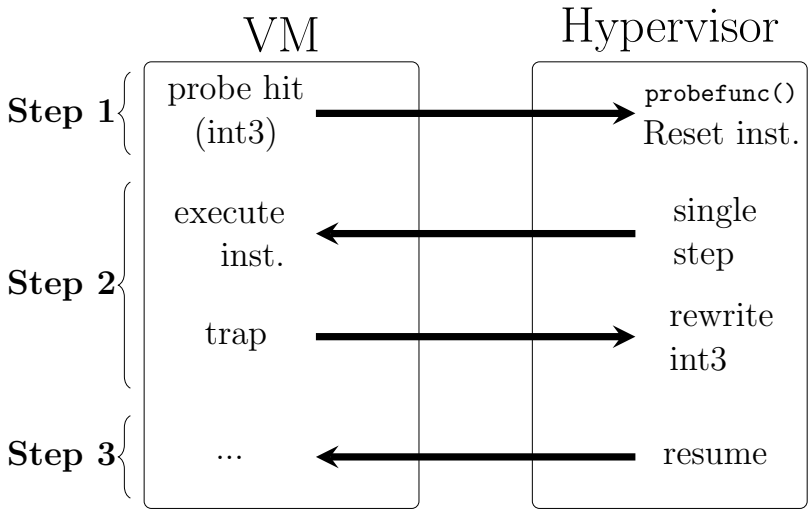


Figure 4.2: A probe hit in the hprobe prototype. Right-facing arrows are VM Exits and left-facing arrows are VM Entries. When `int3` is executed, the hypervisor takes control. The hypervisor optionally executes a probe handler (`probefunc()`) and places the CPU into single-step mode. It then executes 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. © 2015 IEEE

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). We chose `int3` since it is well supported and has a single-byte opcode.[4]

4.2.2 Sample API

In accordance with the simplicity requirement, the API for building detectors using the hprobe framework is simple. The C API for adding and removing probes detectors is shown below:

[4]A single-byte opcode is desired to avoid edge cases such as when the target instruction is the last byte on a boundary. If a multi-byte opcode were used one could overwrite a single-byte instruction with multiple bytes, which is only acceptable when those two instructions are guaranteed to be executed in sequence.

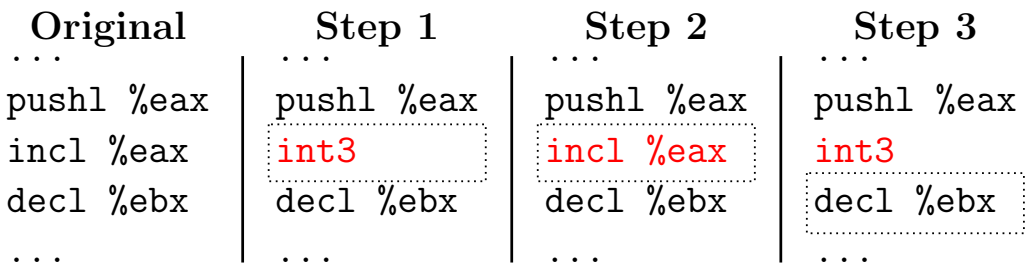


Figure 4.3: Assembly pseudocode demonstrating what an hprobe looks like in the VM's memory before adding a probe (left frame) and during a probe hit (right three frames). The dashed box indicates the VM's current instruction being executed and the red text highlights the probe target as it is overwritten and rewritten. © 2015 IEEE

```
int HPROBE_add_probe(addr_info virt_addr, vmid,
                (*)probe_func(probe_arg *arg));
int HPROBE_remove_probe(addr_info virt_addr, vmid);
```

The `addr_info` structure contains the guest virtual address of the probe location as well as the `CR3` address for user space probes (described in the next subsection), `vmid` is a unique identifier for the target VM (we currently use the process id or pid of the qemu-kvm process for the target VM), and `probe_func` is a function pointer to the (optional) probe handler that has an argument for a structure that contains the vCPU state and other information (e.g., probe address) needed by the handler function. Note that `probe_func` functions are executed in a manner that is safe against manipulation of hprobe state, meaning that one can add and remove probes (including the owner of the currently executing `probe_func`) as a part of handling a probe hit - functionality that will be useful for dynamic monitoring capabilities.

4.2.3 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 kernelspace and guest userspace. That is because while the OS always maps the kernelspace pages at the same address for all virtual address spaces, each user program has its own virtual address space and therefore its own set of page tables. User space probes require the Page

Directory Base Address (from the `CR3` register) to translate a guest virtual address (GVA) into a guest physical address (GPA). Once the GPA is known, we can overwrite the instruction at that address and insert probes into the address space of a particular process. However, as discussed in Chapter 2, 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. In this chapter, we use libVMI to obtain the value of the CR3 register corresponding to the target process's virtual address space [35]. This allows us to translate the virtual address of a probe location (which can be obtained from dynamic or static analysis, (e.g., disassembling the application binary or inspecting the application's symbol table) to a GPA that can be used to add a probe.

If one wishes to add a probe to a userspace application, however, there exists another challenge beyond locating the corresponding `CR3` pointing to the virtual address space belonging to the running process of the application. Unlike the guest OS, the code pages of a running application are not guaranteed to be resident in memory during the lifetime of the application. Essentially, some of the application's code may reside on disk because it either has not been loaded or has been swapped out. When execution reaches a page that exists but is not resident, a page fault will occur and the OS will bring that page into memory. However, this means that the hypervisor may not be able to insert probes directly into all locations of the program at all times (i.e., without an existing translation for a page the host cannot write to that particular GVA in memory). This situation arises particularly during application startup. After loading an application, 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 intended 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 always get called on the next invocation of the instruction at the probe's desired location - even when an hprobe is added before the application accesses the intended probe's location.

One approach to solving the problem of the target guest code page residing

on disk is to wait until the OS naturally brings the necessary page into memory (i.e., the guest OS handles a page fault for that page). As mentioned in Section 2.4, recent versions of x86 Hardware Assisted Virtualization (HAV) use two-dimensional page tables. With TDP, a VM Exit does not occur for all guest OS page table updates. However, it is possible to trap a guest OS page table update using TDP. With Intel's EPT, one can remove access permissions (EPT supports read, write, and fetch/execute permissions) from EPT entries to induce an `EPT_VIOLATION` VM Exit event when those pages are accessed. To detect guest OS page table updates, we remove EPT write permissions in the EPT entry corresponding to the *guest physical page* that represents the ***guest page table entry*** that holds the translation for the *guest virtual page* containing the intended probe's virtual address. At the time of making this EPT change the guest physical page for the desired probe location is not yet present in the guest OS, and therefore not only does a translation from *guest virtual address* to *guest physical address* not exist in the guest OS paging structures, but the data itself is not present in memory. When an `EPT_violation` corresponding to the now write-protected ***guest page table entry*** occurs (indicating that the page containing the probe location has been brought into memory by the guest OS), the CPU is placed into single-step mode. After single-stepping through the instruction that writes the guest page table entry,[5] the probe is added by performing the usual translations from GVA to GPA and traversing the guest paging structures. This process of using page protection to insert probes into non-resident locations is illustrated in Fig. 4.4. Note that the performance of this technique could be improved slightly by avoiding the single-step and decoding the trapped instruction that caused the `EPT_VIOLATION` to obtain the value of the instruction operand corresponding to the page table write.[6] In practice with modern systems containing gigabytes of memory, however, this situation where an application page resides on disk only occurs after application startup (unless a page is swapped out due to memory pressure).

[5]Since the guest page table entry was EPT write protected, a VM Exit occurred and the instruction writing the guest PTE was not executed. The guest PTE is not written until the hypervisor removes write protection and allows the VM to continue executing the instruction writing the guest PTE. A single step is used to ensure that the hypervisor adds the probe at the very first time the translation for the intended probe location exists.

[6]Note that this PTE value is not readily available from the EPT violation; only the guest physical and virtual address of the `EPT_VIOLATION` are available from the VMCS.

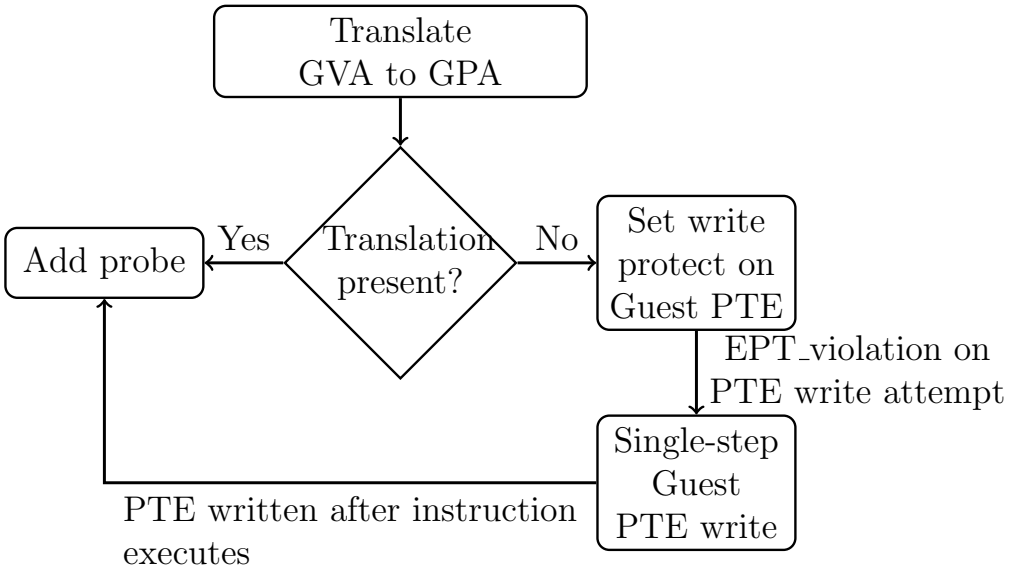


Figure 4.4: How a user space probe is added when accounting for potentially paged-out probe locations. A guest virtual address (GVA) for the probe's location must be translated into a guest physical address (GPA). If the translation fails because the page is not present, we write protect the EPT page containing the guest page table entry (PTE) that should contain the translation for the probe's GVA. When the guest OS attempts to update the guest page table entry, the hprobe kernel agent is notified via an `EPT_VIOLATION` and places the CPU in single step mode. After the single step, the guest PTE is updated and the translation succeeds, and the probe is added. © 2015 IEEE

In either case the disk latency to bring the page into memory would dominate the additional latency from a VM Exit and the performance gain would be negligible. Decoding the guest instruction therefore becomes undesirable due to the complexity of addressing modes in Intel x86.

When performing reliability and security 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 at the time of those events (e.g., registers, flags). 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 in addition to KVM functions to learn additional information about the target VM or possibly modify the state of the VM to mitigate a failure or vulnerability.

4.2.4 Discussion

The use of `int3` to generate an exception means that the hprobe framework's hook event generation is supported and enforced by the hardware: aside from assuming the demand paging scheme with guest userspace probes, the hprobe framework has no dependence on any functionality inside the guest OS. By using only HAV hypervisor functionality the hprobe framework can be used for any guest OS supported by the hypervisor. Since the majority of the work is done outside of direct modifications to the KVM hypervisor (i.e., all of the heavy lifting is done inside of the separate hprobe kernel agent), the framework can also be ported to other hypervisors that support trapping on `int3`.

The hprobe framework satisfies the design principles for an RSaaS hypervisor hook-based monitoring system set forth in Section 4.1.1:

1. **Protection**: By using an out-of-VM approach that is enforced by HAV, hprobes cannot be circumvented. Furthermore, memory protection in the hypervisor can prevent probes from being modified.

2. **Simplicity**: Modifications to introduce the Event Forwarder and Helper APIs to KVM add only 117 source-lines-of-code (SLOC) to KVM's codebase (the base KVM kernel module has approximately 45k SLOC total) 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 boilerplate template (e.g., subsequent detectors can be built by reusing a majority of the code from a previous detector since a majority of the code is module initialization and parameter parsing, unrelated to hprobe handling). As an anecdotal example, most of the example detectors presented in Chapter 5 required only two hours of programming to be fully functional. Hprobes can be used on a completely unmodified guest OS.

3. **Dynamism**: The hprobe 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 at the time of writing, the hprobe framework supports application level monitoring through userspace probes.

4. **Performance**: Multiple VM Exits are required on each probe hit, and one must take that into account when designing hprobe monitors. That said, for the test applications and use cases, the performance is acceptable and worth the value added in terms of implementation simplicity and dynamism. See Section 4.3.2 for analysis and details.

Further elaborating on the protection principle, the hprobe framework satisfies the protection requirements adapted from Lares [5] in Section 4.1.1 and Fig. 3.4. The notification N is only delivered if events occur legitimately (non-hprobe related `int3` exceptions are ignored by the hprobe kernel agent and passed back to the VM). The context and information related to the event (the VM's state at event e) cannot be modified during hprobe processing since the hypervisor is in control and blocks vCPU execution until the single step event. The security application (e.g., a `probe_func()`) 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 envision a denial-of-service (DOS) attack based on forcing VM Exits using many spurious `int3` instructions. However, hprobes do not present a new DOS threat in the KVM hypervisor. If an attacker were interested in performing a DOS attack by introducing a large number of VM Exits, he or she already has the functionality to do so (e.g., using the `vmcall` instruction to invoke a large number of non-existent hypercalls). Note that even if this DOS attack were to succeed, it would not affect the hypervisor's ability to manage its VMs. In fact, the microbenchmarks used to test hprobes in Section 4.3.2 effectively spam the hypervisor with hprobe events. At no point during those benchmarks did the hypervisor even stop responding, the only impact of the VM Exits is the reduction of the useful work that can be completed by the VM (though if an attacker already had access to run arbitrary code on the VM and the goal of the attack was to slow down the VM, he/she could just generate I/O activity).

While using the hprobe framework does require modifications to the hypervisor, these modifications are small in SLOC and robust across multiple versions of KVM and the Linux kernel. During the course of this project, we used the diff-match-patch libraries[7] to migrate the Event Forwarder and

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

Helper APIs between KVM versions. We have tested hprobes on OpenSUSE 11.2, CENTOS7, Gentoo with kernel version 3.18.7, Ubuntu 12.04 using kernel versions 3.13.0-30-generic and 3.13.0-45-generic, and Ubuntu 14.04. The hprobe kernel agent is written to be agnostic to kernel versions starting with 2.6 (e.g., `#ifdef` macros for kernel version specific constructs like `unlocked_ioctl` were used). To help ensure that our modifications do not affect the robustness of the hypervisor, we confirmed that the KVM unit tests[8] pass on our modified version of KVM.

4.2.5 Limitations

The hprobe framework can be used as a key component in an RSaaS framework, but 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., to be alerted every time a particular address is read from/written to), hprobes do not provide a clean way to do so. In order to monitor data reads/writes only using hprobes, 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 [3], but with the dynamism of hprobes. This concept is used in the RSaaS ret2user detector presented in Chapter 8.

2. Hprobes are built on HAV and hprobe invocations involve multiple VM Exits, resulting in higher performance overhead than other approaches (e.g., SIM [7]). However, this tradeoff was chosen to have a simpler and more robust implementation with its trust still rooted in HAV. A key design decision behind RSaaS is that the monitoring system must leave the guest OS unmodified.

3. Probes cannot be fully hidden from the VM. Even with clever EPT tricks to hide the existence of a probe when reading from a page containing a probe, a timing side channel would still exist since an attacker

[8] http://www.linux-kvm.org/page/KVM-unit-tests

could observe that either the probed instruction takes longer than expected to complete or memory reads of a code page containing hidden probe take longer than expected.

4.3 Hprobe Performance

4.3.1 Methodology

All of the microbenchmarks described in this section as well as the sample detector performance evaluations in Chapter 5 were conducted on a Dell PowerEdge R720 server with dual-socket Intel Xeon E5-2660 "Sandy Bridge" 2.20 GHz CPUs (3.0 GHz turbo boost).

Since a virtual machine does not have complete control over the hardware, some care should be taken to ensure accurate measurements when trying to obtain timing measurements, especially for low latency events. One particular problem for performance measurements is unstable timekeeping inside the guest OS [74]. Since a VM timeshares with the host OS, there will be more clock drift than expected for a physical server (e.g., when a vCPU gets preempted). Also, one could expect clock drift to be more unpredictable as it is dependent on the state of a live system (e.g. how applications running on the hypervisor and other VMs use the system) compounded with expected sources of clock instability (temperature fluctuations, etc). In the case of VM benchmarks, there may also be VM Exits induced by other background activities on the system. While there are well established techniques for minimizing the effect of virtualization on accurate timekeeping (e.g. a paravirtualized clock synchronized with the host, **kvm-clock**), precise timing measurements in VMs can be difficult to obtain. For precise measurements, it is considered best practice to use a stable reference outside of the VM [75]. In these experiments, we measure time from the host OS. To obtain runtime measurements of activities inside the VM, an extra hypercall was added to KVM. The hypercall is used to start and stop a timer inside the host OS. This hypercall technique allows us to obtain more consistent measurements than inside a VM with a potentially unstable clock. To ensure consistency among measurements, the test VMs were rebooted between each sample.

4.3.2 Microbenchmarks

In order to determine a lower bound for the overhead of hprobe-based detectors, the base of overhead of hprobes must be quantified. This section contains information and results from running hprobe microbenchmarks, that is hprobes without a probe handler function. A microbenchmark aims to 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. 4.2). The round-trip latency of an individual VM Exit on Sandy Bridge CPUs has been estimated to take roughly 290 ns [76] and the hypercall measurement scheme induces additional VM Exits; it would be difficult to accurately measure individual probe latency. Instead, a mean round-trip latency is obtained by repeatedly executing a probed function many times (one million) and dividing the total time taken for those executions by the number of executions. This aggregate measurement of the mean helps remove jitter due to timer inaccuracies as well as ensure that the actual latency of the hypercall measurement does not affect the result. The test probe location was a no-op kernel module that was added to the Guest OS. The kernel module creates a dummy `noop` device with an `ioctl` that calls a `noop_func()` kernel function that performs no useful work (it only includes a `return 0`). First, an hprobe is added to the `noop_func()`'s location. The microbenchmarking process 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, a 32bit Ubuntu 14.04 guest was used and 1000 samples were measured (that is, a total of 10^9 hprobe invocations). The mean latency (across samples) was found to be 2.6 μs. In addition to the Sandy Bridge CPU, the experiment was repeated on an older generation 2.66 GHz Xeon E5430 "Harpertown" processor (running the same kernel, KVM version, and a copy of the VM image). This CPU generation was chosen since it is from the same timeframe as other hook-based monitoring systems that will be used for comparison in the next section. The Harpertown CPU had a mean latency of 4.1 μs. The distribution of latencies for these experiments is shown in Fig. 4.5.

The hprobe prototype requires multiple VM Exits per probe hit and thus

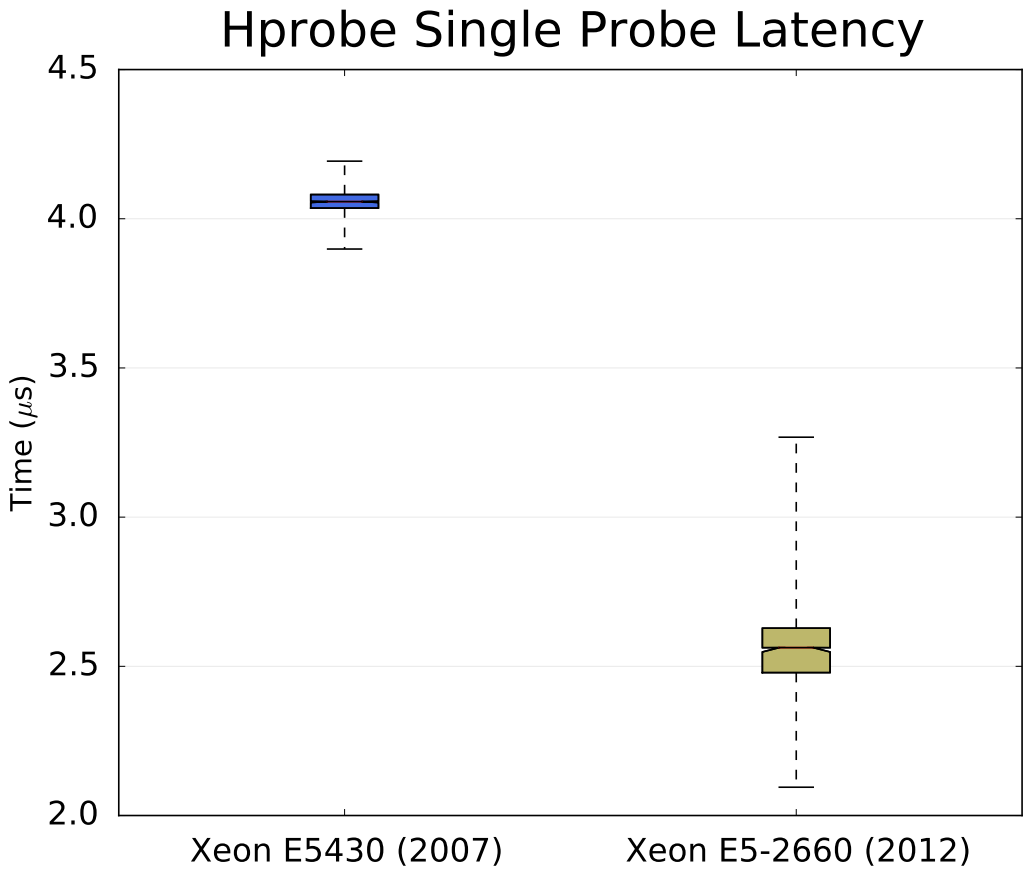


Figure 4.5: Latency of a single hprobe (parentheses indicate each CPU's release year). The E5-2660's larger range can be attributed to "Turbo Boost," where the clock scales from 2.2 to 3.0 GHz. The shaded area is the quartile range (25^{th} percentile to 75^{th} percentile), whiskers are minimum/maximum, center is the mean, and notches in the middle are the 95% confidence interval of the mean. © 2015 IEEE

a fair amount of time is needed to process each hprobe. 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 in Chapter 5. 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 [76].

4.4 Comparison to Past Hook-based Techniques

In the hprobe prototype implementation, the mean latency for a single hprobe was 2.6 μs (4.1 μs for a CPU from 2007). This fits between the single-hook latency of Lares (28 μs, published in 2008) and SIM (0.4 μs, published in 2009). While the raw overhead of a probe hit may appear relatively high, it is important to remember that probe events are rare and even the sample application benchmarks in Chapter 5 intentionally exercised the probes beyond what may be typical for a real application. The dynamism of the hprobe framework allows hprobe-based detectors to remove probes as needs change, allowing an RSaaS system to disable detectors that are no longer needed at runtime. It is difficult to achieve a similar flexibility with techniques that have hooks statically inserted into the guest OS. In other hook-based systems, in order to provide flexibility when guest OS modifications are required, one would either have to maintain a set of guest OS kernels or use a scheme that modifies a running kernel image from within the guest at runtime.

The hprobe hooking mechanism is enforced by the hardware since it is supported by the VM Exit mechanism. As mentioned in Section 4.2.4, the hprobe framework's use of `int3` with the `VMCS` exception bitmap configured accordingly can be viewed as a system rooted in hardware architectural invariants [8, 43]. In this case, the invariant is that a properly functioning virtual machine will generate VM Exits on exceptions that it is configured to VM exit on (an assumption that is essential for a "trap-and-emulate" VMM). To protect the hooks, one can use EPT or NPT and write protect the pages that contain active probes. This write protection satisfies the security requirement where hooks cannot be evaded by actors inside the VM (even attackers with guest OS privileges) and only incurs a performance impact when pages containing probes are written to (a rare event for code in memory, particularly guest OS code).

Quynh et al. utilized `int3` to trap from a VM to the hypervisor in xenprobes [77], which provides a guest OS kernel debugging interface for Xen VMs. However, hprobes were developed with a focus on reliability and security monitoring as opposed to debugging use cases. Additionally, xenprobes can use an out-of-line execution area (OEA) to execute the replaced instruction (vs. always executing in place with a single step like the hprobe prototype does). The OEA provides a performance boost, but it results in a more

complex code base and carries the need to create and maintain a separate memory region for this area. The OEA requires an OS driver to allocate and configure the OEA at guest OS boot (increasing the burden for a RSaaS system), and the OEAs are statically allocated at boot, placing a hard upper bound on the number of supported probes (which is likely acceptable for debugging, but not for dynamic reliability and security monitoring). In terms of code complexity, the hprobe framework is simpler (less than 1000 lines of code vs. 4000 lines of code) and likely yields a smaller attack surface since no guest code is executed by hypervisor (which is not the case or a concern in the OEA for the xenprobes debugging system).

Chapter 5

Sample Hprobe Detectors

In this chapter, we present sample reliability and security detectors built upon the hprobe prototype framework. While these detectors are only examples, they are unique to the hprobe framework and cannot be implemented on any other VM monitoring system at the time of writing. As in Chapter 4, the benchmarks presented below use the hypercall timing system and the Sandy Bridge E5-2660 based Dell PowerEdge R720.

5.1 Application Heartbeat Detector

5.1.1 Application Heartbeat Detector Design

One of the most standard and most used reliability techniques used to monitor computing system liveness is a heartbeat detector. In a heartbeat detector, a periodic signal is sent to an external monitor to indicate that the system is functioning properly. Many networked applications have a form of built-in heartbeat or timer, but applications that do not have a persistent network connection to another host may not have any liveness-checking mechanism like a heartbeat (e.g., a standalone scientific computing application). Furthermore, network-based heartbeats are prone to false positives caused by failure of the monitoring infrastructure (e.g., a dropped packet or network fault). Even if an application does have a general heartbeat, one may be interested in monitoring a particular component/function of that application (either because it is known to fail or because it may be considered more critical). A heartbeat detector 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 exe-

cution 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 operation of the hprobe-based application heartbeat detector is illustrated in Fig 5.1. This sample detector is implemented as a kernel module that is installed in the Host OS (i.e., one of the detectors on the left side of Fig. 4.1). In the application heartbeat detector, an hprobe is inserted at the start of the code block that is expected to be periodically reentered (e.g., the main loop of an application). When the hprobe is inserted, a delayed workqueue[1] is scheduled to execute at a time in the future corresponding to the expected reentry period for the code block. The workqueue function alerts the host OS that a heartbeat has been missed. When the timeout expires, the workqueue function is executed and declares the guest application has failed. If the user desires a more aggressive watchdog detector, one could have the hprobe handler perform an action such as restarting the application or VM. During the application's expected execution (i.e., when the hprobe gets executed), the workqueue is canceled and a new workqueue is scheduled for the same interval, starting a new timeout period. In this fashion, application failure is detected whenever the time since the hprobe was invoked is greater than the predetermined timeout. This scheduling, probe invocation, and cancelling of the workqueue continues until the application finishes or the user no longer desires to monitor that application and removes the hprobe. If having an hprobe hit on every iteration of the main loop is too costly, one can use a dynamic approach. In the dynamic monitoring approach, ensure that the probe is 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). Dynamic monitoring for failure detection is revisited in the context of guest OS hang detection in Chapter 7.

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

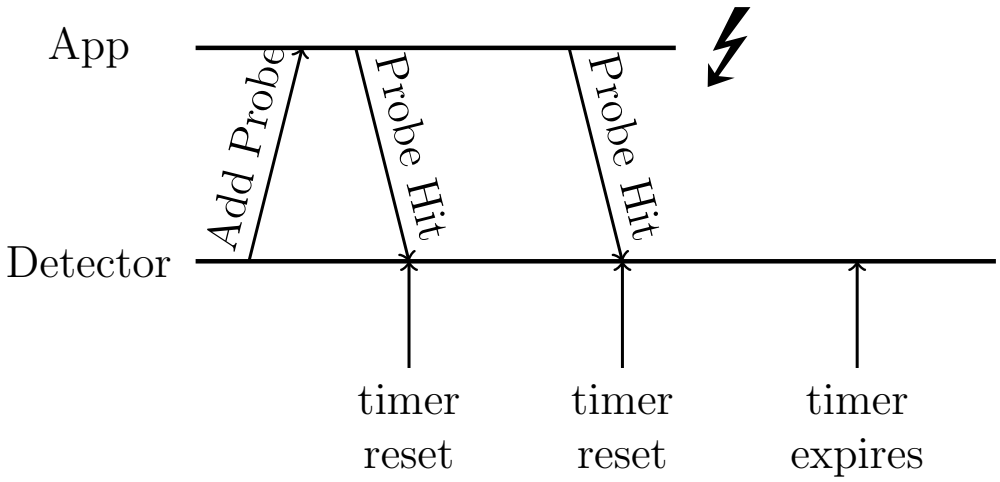


Figure 5.1: Basic operation of the Application Heartbeat Detector. A probe is added to a critical section of the application that is executed periodically (e.g., the main loop). During normal execution, a workqueue in the host OS is reset whenever the probe is hit. In the presence of a failure (such as an I/O hang due to a disconnected device), the workqueue expires and executes a function to notify the host OS. © 2015 IEEE

5.1.2 Application Heartbeat Functional Evaluation

To test the application watchdog, we need to find an application that is intended to run for a long time and does not have a built-in fault tolerance mechanism. A good example application is a long-running scientific program that can take many hours or days to complete. This is useful for an as-a-service approach since many authors of scientific applications would like to focus on application development and not on details such as fault tolerance. We use the open-source Path Integral Quantum Monte Carlo (pi-qmc) simulator [78] as a test application.[2] As is typical with scientific computing applications, pi-qmc has a large main loop that is repeatedly executed over the lifetime of the application. When profiling to determine the heartbeat timeout, we only need to run the main loop a handful of times to determine the maximum expected time per iteration. This is possible since Monte Carlo simulation involves repeated sampling and therefore repeated execution of the same functions. Furthermore, sources of non-determinism are rare since pi-qmc is heavily CPU bound. After determining the expected duration of each iteration, we set the heartbeat to timeout to twice the ex-

[2]available at: http://phys-tools.github.com/pi-qmc/

pected value, set the detector to a statement at the end of the main loop, and injected hangs (e.g., by sending SIGSTOP from within the guest OS) and crashed the application (e.g., by sending SIGKILL in the guest OS). In addition to interfering with the application's execution, we also injected VM suspends and crashes using libvirt. All crashes (including VM crashes since the timer executes in the hypervisor) were detected.

5.1.3 Application Heartbeat Detector Performance Evaluation

For benchmarking the application heartbeat detector, we use the same pi-qmc simulator from the previous section. 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. We measure both with and without the heartbeat active and run the program for 15 minutes. The results of these experiments are shown in Fig. 5.2.

From Fig. 5.2, we can see 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, making this loop a good choice for a probe location. Furthermore, by setting the threshold of the detector to a conservative value (such as twice the mean runtime), one can achieve fault detection in a far more acceptable timeframe than other methods like manual inspection of running compute jobs. Also, 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). The fundamental principle of this detector is that we detect failure by directly monitoring the application's execution.

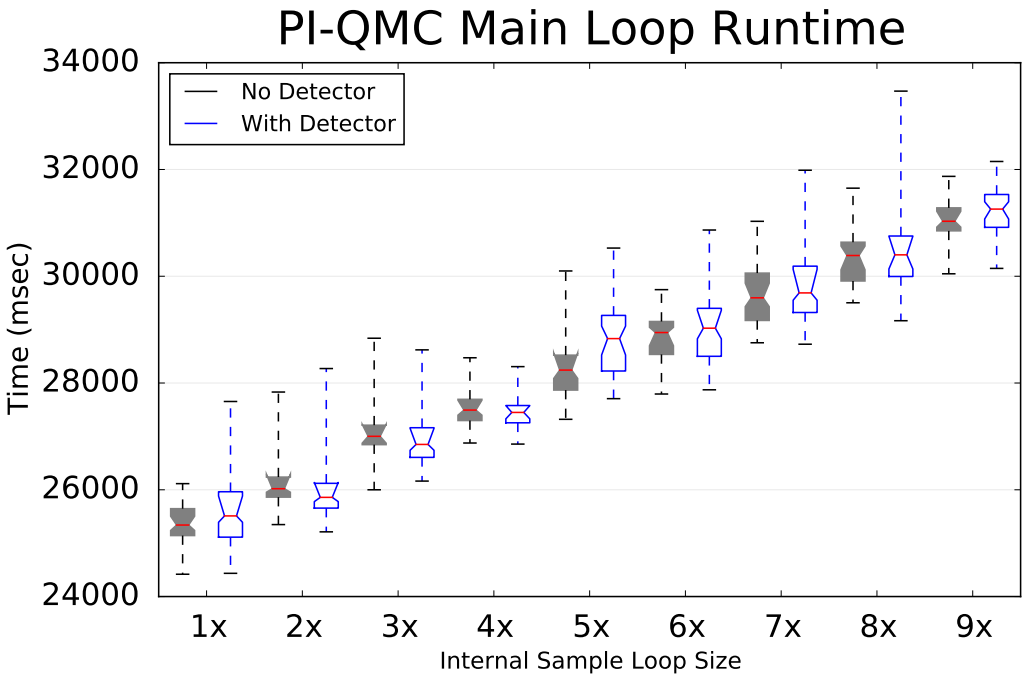


Figure 5.2: Benchmarking of the application watchdog detector using the main loop in the pi-qmc simulator. The horizontal axis indicates the scaling of an internal loop in pi-qmc. The vertical axis shows a distribution of the completion time for each iteration of the main loop. The boxplot characteristics (e.g., quartile range) are the same as in Fig. 4.5. © 2015 IEEE

5.2 Infinite Loop Detector

5.2.1 Infinite Loop Detector Design

As anyone who has interacted with software knows, infinite loops are a common failure that can cause application hangs. When considering proper execution of a loop in a program (that is likely not the main loop as in the Monte Carlo example), the number of instructions executed in a given block of code usually falls into a fixed range. The upper bound of the instructions executed in a block is the worst case execution time (WCET) [79]. Determining the WCET is a well studied problem in real-time systems, and solving it is beyond the scope of this work. For example, one can use an automated system to infer loop invariants and bound the number of times a loop should execute [80]. This allows one to measure the WCET in terms of a higher level construct (e.g., the number of loop iterations). The infinite loop detector presented below does not apply only to loops: if one can identify a block of code or function that is expected to be executed repeatedly, the number of times that block is executed before the end is reached should also fall into a fixed range.

Given a block of instructions representing the interior of a loop and the WCET (either in units of time or the number of executions of that block), one can build a detector using a pair of hprobes. When one knows the wall clock time, one can insert an hprobe inside the block and another hprobe after the block. This two-probe detector is illustrated in Fig. 5.3. At the first probe, a timer is started (using the same technique as the heartbeat detector in Section 5.1). 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 [8, 43] to only count the time when the application under consideration is being executed. This application-specific time monitoring can be accomplished 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 [81]). Depending on

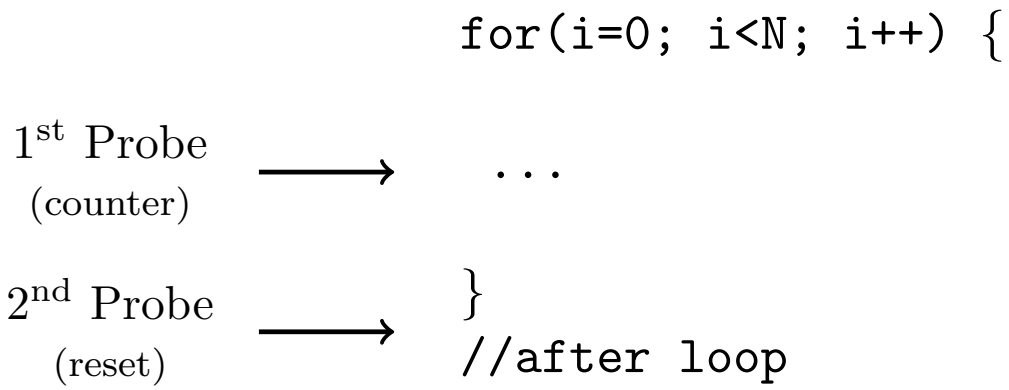


Figure 5.3: Probe locations for the infinite loop detector (ILD). The ILD has two modes of operation, both utilizing the same set of probes. In the first mode, failure is declared when the loop executes more times than a set threshold. The second mode of operation tracks the register state. If more than a specified set of registers remains static for N iterations, failure is declared. © 2015 IEEE

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 for bounding the maximum number of loop iterations, one can also detect an infinite loop by observing the state of the CPU [82]. If the register state remains static across many iterations, one can infer that the application is hanging. The register state of the VM is saved in the VMCS upon a VM Exit, and KVM stores this state in its own structures to be reloaded immediately prior to the next VM entry. As mentioned in Chapter 4, the hprobe detectors have access to this state upon probe activation. Therefore, the infinite loop detector can check this state at every loop iteration. If a thresholded number of general-purpose registers[3] remain constant across a number of iterations greater than a threshold, one can infer that the application is in an infinite loop. This register tracking strategy can be combined with WCET techniques since the register state may help identify whether or not the system is looping or providing useful work.

[3]The threshold will usually be 1, as a register will often contain a loop iterator variable. The loop iterator can easily be identified because it will be a register that is either monotonically increasing or decreasing by one at each iteration

5.2.2 Infinite Loop Detector Functional Evaluation

In order to test the infinite loop detector, we used the same example as presented in Jolt [82]. That example is a bug found in a development branch of the Exuberant Ctags source code indexer.[4] In that bug, a string parsing loop for interpreting Python strings 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.[5] Specifically, the `_import_tools.py` file contains comments that are formatted in such a way that the bug is exercised. In the fixed version of the code the loop executes only one iteration each of the twelve times the loop is entered, meaning a small threshold timeout could be used. Using both the threshold and register state method, the infinite loop was easily detected in all experiments since it executes at a rate of thousands of times per second.

5.2.3 Infinite Loop Detector Performance Evaluation

In order to measure the performance overhead of our infinite loop detector, we use a patched version of the ctags application from Section 5.2.2. We ran ctags on the entire numpy source tree 60 times and obtained the mean runtime and 95% confidence interval. The results are tabulated in Table 5.1. In these experiments, we utilized two implementations of the detector: a "Naïve" detector and a "Smart" detector. The Naïve detector is the same detector as presented in Section 5.2.2 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 (i.e., the VM was rebooted before collecting each sample). The rows in Table 5.1 with "Page fix" refer to the runs where the application was started for the first time and we needed to use the EPT mechanism presented in Section 4.2.3. We also ran experiments in which we forced the application to page in the code page containing the target instruction at startup, represented by the "No Page Fix" samples.

From Table 5.1 we can see that the performance impact of our solution

[4] http://ctags.sourceforge.net/
[5] http://www.numpy.org

Table 5.1: Ctags on Numpy Source Tree © 2015 IEEE

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

to deal with paged-out user space application code is not statistically significant (compare the "Page Fix" rows of the same detector to the "No page fix" rows). However, using a dynamic probing technique yields measurable performance gains. In the Naïve approach, the overall overhead is roughly 11.5% for these experiments. With the Naïve detector, the first and second probe get executed 2585 and 54308 times, respectively. The disparity in execution of the first and second probe is due to the fact that in this application the loop is often skipped over, but the instruction immediately after the loop (i.e., what the second probe measures) is always executed. In the Smart approach, the first and second probe both get executed 2585 times (as a reminder, in correct operation of ctags for the numpy source tree the loop has only one iteration so the second probe is added and executed only once whenever the first probe is reached), yielding a nominal difference of 1-2% between the Smart implementations and the base case without any probes. If this loop had a large number of internal iterations, then one could use a similar dynamic probe approach, but retain the exit (second) probe and remove the internal (first) probe, adding it periodically or using a timeout mechanism. This idea of removing and re-adding probes for increased performance is explored further in the OS hang detection method discussed in Chapter 7. Note that the capability behind the "Smart" approach is unique to the dynamism in the hprobe framework and to the best of our knowledge no other VM monitoring framework offers such capability.

5.3 Emergency Exploit Detector

5.3.1 Emergency Exploit Detector Design

Most systems operators fear zero-day vulnerabilities as there is little that can be done about those vulnerabilities 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 before deployment into a production environment. This can further be exacerbated in environments with high availability concerns and stringent change control requirements: even if a patch is available, it is often not possible to restart the system or service until a regular maintenance window. This leaves operators with a difficult decision: either risk damage from restarting a system with an under-tested patch or risk damage from running an unpatched system.

Consider the CVE-2008-0600 vulnerability that resulted in a local root exploit through the `vmsplice()` system call [83, 84]. This example represents a highly dangerous buffer overflow since a successful exploit allows one to arbitrarily execute code in ring 0. Furthermore, there is a publicly available program on the Internet for starting a shell with root privileges that exploits the CVE-2008-0600 vulnerability. Since this exploit involves the base kernel code (i.e., not a loadable module or driver), patching it would require installing a new kernel followed by a system reboot (or without a reboot using techniques discussed at the end of this section). 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()` exploit constructs specially crafted compound page structures in userspace that can be used to trick the kernel into executing an arbitrary function. 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 the underlying pages in the compound page. The exploit works by using an integer overflow to corrupt the kernel stack and replace the kernel's page structure references with references to the

specially crafted compound pages in userspace. Before calling `vmsplice()`, the exploit closes the pipe. Then `vmsplice()` is called and the compound pages' destructor function is invoked. This 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 integer value of `iov_len` and is set to `ULONG_MAX` in the exploit. Since Linux uses registers to hold the system call number as well as arguments for system calls [85], we could use classical system call monitoring/tracing tools to detect this exploit [86–88]. In this system call tracking approach, we can watch whenever a system call is invoked and parse arguments to detect an integer overflow attempt whenever `vmsplice()` is invoked. However, since hprobes are dynamic and can be added at runtime, instead of tracking all system calls we can set a probe to trigger only on the `sys_vmsplice()` function (the internal kernel function called after the system call assembly linkage). Adding the probe to `sys_vmsplice()` ensures that only the execution path of the `vmsplice()` system call is inspected as opposed to all system calls (as would be in traditional system call tracing). When `sys_vmsplice()` is executed, the kernel is using the regular compiler function calling conventions (in most versions of the Linux kernel, the gcc convention) and the arguments are stored on the stack. Irrespective of the calling convention, we can use hprobes to obtain these arguments. Pseudocode describing how the detector is implemented is shown in Fig. 5.4. Essentially, one needs to ensure that the value of `iov_len` will not cause an overflow (which should never occur in a benign case since an `iov` cannot occupy the entire virtual address space). Depending on the environment, the operator can choose how to respond to the detected exploit. One could send an alert, simply modify `iov_len` to a benign value that causes `vmsplice()` to fail (e.g., 0 in our testing), or take a more drastic action (such as terminating 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,[6] which checks if the memory region specified by the argument is a valid memory region in userspace. One could write a probe handler that performs a similar check by examining if all of the region referred to by

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

```
1: procedure VMSPLICE_HANDLER(vcpu)
2:     if 32-bit guest then
3:         arg_offset = 8                          ▷ 2nd arg on stack 32bit
4:         max ← ULONG_MAX_32 - PAGE_SIZE
5:     else
6:         arg_offset = 16                         ▷ 2nd arg on stack 64bit
7:         max ← ULONG_MAX_64 - PAGE_SIZE
8:     end if
9:     ▷ The read function checks for a valid address
10:    iov_pointer ← read_guest(vcpu.esp+arg_offset)
11:    iov_len ← read_guest_virt(iov_pointer)
12:    if iov_len ≥ max then
13:        HANDLE_EXPLOIT_ATTEMPT(vcpu)
14:    end if
15: end procedure
```

Figure 5.4: Pseudocode for an hprobe based CVE-2008-0600 Detector. This handler is executed when the `vmsplice()` system call is used. The overflow occurs when a struct member in the second argument is ULONG_MAX. The code protects against the integer overflow by ensuring that if a ULONG_MAX argument that would cause an overflow is used, the exploit is caught.© 2015 IEEE

the `struct iovec` pointer + `iov_len` is valid for the calling process (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 service level agreements).

5.3.2 Emergency Exploit Detector Functional Evaluation

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, the ease of rapidly deploying a stopgap 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 such as killing the VM), 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 were detected.

Ksplice [89], a rebootless kernel patching mechanism, can be used in a similar fashion as the `vmsplice()` emergency detector. Ksplice allows for live

kernel patching by replacing a function call with a jump to a new patched version of that function. The 4.0 version of the Linux kernel was scheduled to incorporate a rebootless patching feature [90]. The planned Linux 4.0 feature used ftrace[7] to switch to a new version of the function after some safety checks. However, those safety checks were still not stable in time for the 4.0 release and the feature is not yet merged into the mainline kernel at the time of writing. In particular, ensuring that the stack is consistent has proven to be exceptionally challenging, with the stack validation checks having been revised 13 times and still not production ready [91, 92]. While these rebootless patching techniques can be useful for patches that have been properly tested and worked through a QA cycle, many operators would be uneasy with an untested patch on a live OS. When considering newly reported vulnerabilities, hprobe's simple interface allows one to quickly deploy an out-of-band monitor to detect the vulnerability without modifying the control flow of a running kernel. Since the monitoring only reads the state of the guest OS and checks are performed outside of the kernel, there is little risk of crashing the system or causing unwarranted side effects. When compared to rebootless patching techniques, there is no concern with how hprobe emergency detectors interact with the guest OS stack. If and when rebootless patching becomes stable, hprobe temporary monitoring could even be used to provide a stopgap measure while a rebootless patch is in QA testing: one could use the monitor immediately after a vulnerability is announced and until the patch is vetted and safe to use. A technique like this would drastically reduce the vulnerable window and alleviate pressure to perform risky maintenance outside of critical windows. It should be noted that while our example focused on a kernel vulnerability, this emergency detector technique can be extended to a user space program.

5.3.3 Emergency Exploit Detector Performance Evaluation

The integer overflow detector that protects against the CVE-2008-0600 `vmsplice()` vulnerability is extremely lightweight for most use cases. 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

[7] http://elinux.org/Ftrace

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 [93], we believe this low-overhead should extend beyond the `vmsplice()` example.

One application that uses `vmsplice()` is Checkpoint/Restart in Userspace (CRIU).[8] CRIU uses `vmsplice()` to capture the state of open file descriptors referring to pipes. We used the Folding@Home molecular dynamics simulator [94] and the pi-qmc Monte Carlo simulator as test programs. We ran these applications in a 64-bit Ubuntu 14.04 VM. For each sample, we allowed the application to warm up (load input data and start the main simulation) and then initiated a checkpoint using CRIU. Since CRIU is an open source application, we inserted timing hypercalls directly into CRIU to measure how long it takes to save the checkpoint of an application. The checkpoint experiments were repeated 100 times for each case with and without the detector and the results are tabulated in Table 5.2. 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 when taking a checkpoint of Folding@Home. When checkpointing Folding@Home, `sys_vmsplice()` was called 28 times, whereas `sys_vmsplice` was called 11 times for pi-qmc. We attribute this performance difference not only to probe latency, but also to the negative cache effects of the VM/Host OS context switch when activating probes. We also measured a class of "Naïve" detector that probes the `system_call()` function (the entry point for all system calls) as opposed to `sys_vmsplice()`. This naïve detector checks the system call number and then the arguments to ensure that the integer overflow will not occur. In the naïve 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 from $\sim$ 10-30 to $\sim$ 3000). We remind the reader that the detector only probes `sys_vmsplice()`, meaning that the overhead in this experiment is only incurred when taking a checkpoint and the actual execution of the scientific applications is not affected.

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

Table 5.2: CVE-2008-0600 Detector Performance Evaluation using Checkpoint/Restart In Userspace © 2015 IEEE

Application	**Runtime** ± 95% CI (s)	**overhead (%)**
F@H Normal	0.221 ± 0.0092	0
F@H w/hprobe	0.228 ± 0.012	3.30
F@H w/Naïve	0.253 ± 0.0085	14.4
pi-qmc Normal	0.137 ± 0.0063	0
pi-qmc w/hprobe	0.140 ± 0.0073	1.73
pi-qmc w/Naïve	0.152 ± 0.0051	11.1

Chapter 6

Using OS Constructs to Bypass the Semantic Gap

6.1 Purpose and Motivation

The "as a service" computing model can be characterized by many traits [95], but the ones most useful for reliability and security are on-demand capabilities and increased automation. In order to provide reliability and security monitors with those functionalities, we need a platform that allows us to change how we monitor a running system without affecting that system, while allowing us to have complete access to the system's state (user and OS address spaces, hardware, etc.). VM monitoring can satisfy both of these requirements since the hypervisor has complete control of and access to the underlying guest OS.

Traditional VM monitoring methods (i.e., VMI [3,35]) rely on specific guest OS offsets and addresses to decode VM data structures. As such, VMI-based monitors are subject to the semantic gap and must be adapted for every target version of the guest OS. Furthermore, the information decoded by VMI describes the state of the VM, but does not provide much information on what the VM is actually doing at a given time. To avoid the semantic gap, existing VM monitoring research has used privileged hardware events to gain insight into VM operation [6,8,43]. While robust against attackers, those approaches are limited in functionality (i.e., to privileged hardware operations that can generate VM Exit events). To overcome that limited functionality, other event based systems add hooks to the underlying VM. These hooks transfer control to the monitoring program when the VM reaches a point of interest in its execution [5, 53, 77]. The hprobe framework introduced in Chapter 4 is an example of such a hook-based monitoring system. However, hook-based approaches like hprobes reintroduce the need for semantic information (i.e., knowledge of guest OS function addresses such as hook locations). The

approach presented in this chapter provides a novel alternative for dealing with the semantic gap by developing monitors that rely on OS constructs than can be used to infer any necessary semantic information.

Since our technique bypasses the semantic gap and takes advantage of hprobes' ability to adapt monitoring at runtime, we envision these combined techniques being used as the backend in a system where a user is offered reliability and security monitors as a service. As a research implementation, we limit the scope of our RSaaS system to the components that perform monitoring and not the actual user interface that would be part of such a system. The rest of the cloud API and interface (e.g., integration with a framework like OpenStack[1]) are left as future engineering work.

6.2 Approach

We bypass the semantic gap by using OS constructs to infer version-specific monitoring parameters. OS constructs are useful for inferring parameters because while the hypervisor has access to the full state of the guest OS (e.g., memory and registers), the hypervisor does not have an interpretation of what that hardware state represents. OS constructs allow us to identify events of interest for monitoring based on hardware events visible to the hypervisor (e.g., a system call is represented by a `sysenter` instruction). To identify parameter values for a particular guest OS version, we look for changes in the hardware state, whether in the form of an interrupt/exception, a register access, or the execution of a specific instruction. These hardware interactions map directly to core OS functionality and allow us to develop VM monitoring tools that use those parameters without. Our method for translating low-level information available to the hypervisor to high-level information usable for monitoring is illustrated in Fig. 6.1.

The workflow for building a monitor based on our OS construct approach is shown in Fig. 6.2. Starting with a specification for a monitor, we choose an OS construct that can be used to provide information that satisfies the specification. After choosing an OS construct, we identify the monitor's parameters. We define a *parameter* as an aspect of the guest OS needed for hypervisor-level monitoring that is expected to change with the OS version

[1] http://openstack.org

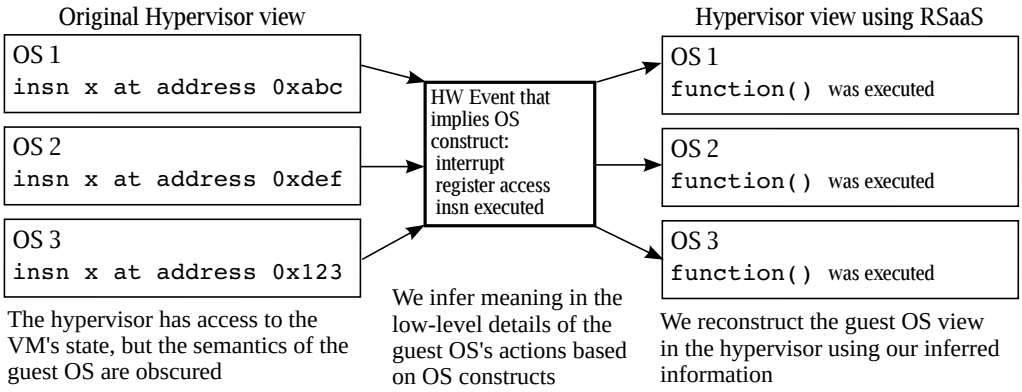


Figure 6.1: Using OS constructs, we are able to assign meaning to the low-level data available to the hypervisor. In the example shown above, we are able to determine that a particular function was called because a special instruction was executed.

(e.g., a function address, interrupt number). The next step is to identify version-specific parameters and then determine a hardware event that is associated with the OS construct used for monitoring. After identifying the hardware event associated with the parameter, one can find the value of the parameter by running the VM and observing that hardware event.

6.2.1 Summary of Guest OS Information Inferred in Example Detectors

A core contribution of this dissertation is to demonstrate how one could build detectors using both hprobes and the techniques presented in this section. The insight behind our observations on how to extract useful information from fundamental OS operations is best demonstrated through examples. The examples presented in Chapters 7-9 are summarized in Table 6.1.

6.3 Prototype

6.3.1 Example Architecture for Reliability and Security as a Service

Integrating our OS construct approach into an RSaaS prototype combines both a Dynamic Analysis Framework (DAF) and a VM Monitoring Frame-

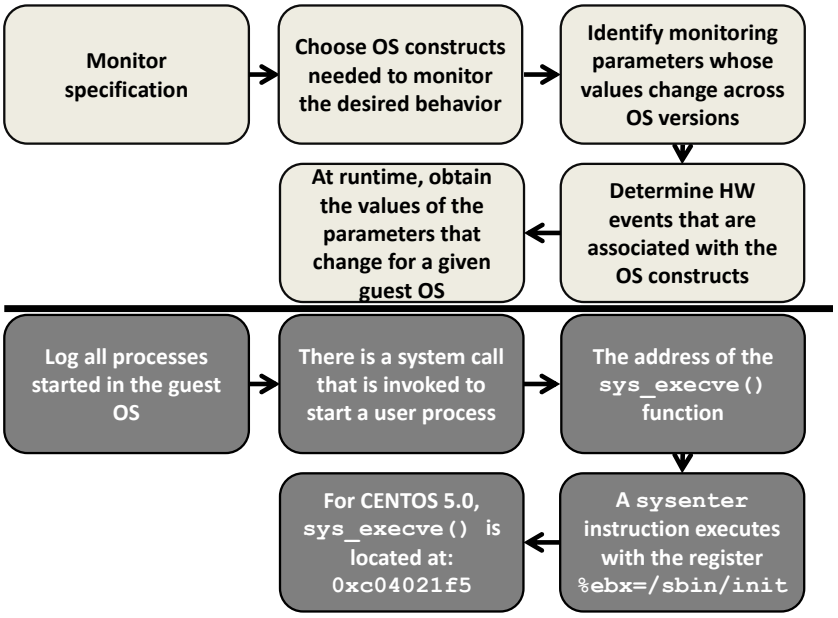


Figure 6.2: The workflow used when building a detector for the Reliability and Security as a Service framework. The light-colored boxes on the top describe the high-level workflow, and the bottom darker-colored boxes show the same workflow for identifying the process creation system call. After this workflow has been completed by the cloud provider, the monitor is ready to be deployed transparently across various customer VM instances.

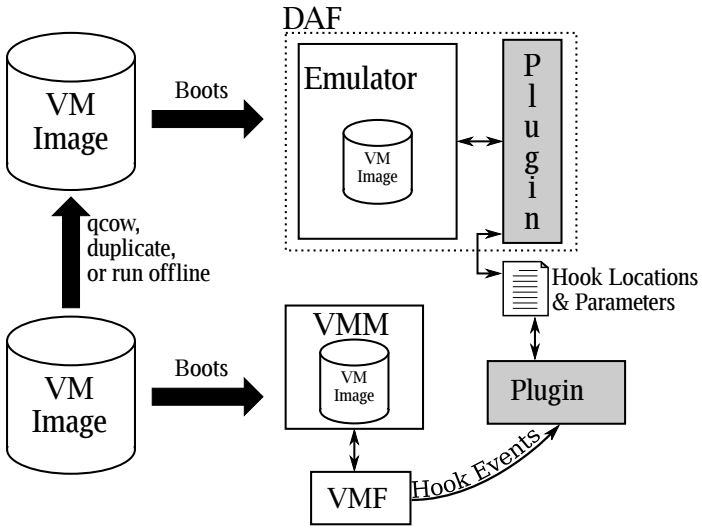


Figure 6.3: Architecture proposed in this work. Each monitor is defined by a Dynamic Analysis Framework (DAF) plugin and a VM Monitoring Framework (VMF) plugin (gray boxes).

Table 6.1: Parameter Inference in Example RSaaS Detectors

Monitor	**OS Construct**	**Parameters**	**How Parameters are Inferred**
OS Hang Detection	During correct OS execution, the scheduler will run periodically.	The scheduler address; the maximum expected time between calls to the scheduler	Find the function that changes process address spaces; record the maximum time observed between process changes when the system is idle.
Return2User Attack Detection	The transition points between userspace and kernelspace are finite; code in userspace is never executed with system permissions.	The addresses for the entry and exit points of the OS kernel	Observe the changes in permission levels and record all addresses at which those changes occur.
Keylogger Detection	A keyboard interrupt will cause processes that handle keyboard input to be scheduled.	Interrupt number for the keystroke; number of processes expected to respond to a keystroke	Use virtual hardware to send a keystroke and record the interrupt number that responds; observe scheduling behavior after a keystroke.

work (VMF). The DAF performs the parameter inference step specific to each cloud user's VM (e.g., finding guest OS system call addresses). The VMF performs the runtime VM monitoring. The architecture for the RSaaS prototype is shown in Fig. 6.3. Note that the proposed approach is quite general and other implementations may choose to use different techniques for parameter inference (e.g., static binary analysis of the guest OS kernel image) or runtime monitoring (e.g., kprobes for containers [71]). The user selects a monitor to deploy to his or her VM and that monitor is represented by two monitoring plugins: one for the DAF and one for the VMF (alternatively, we imagine scenarios where the provider may not wish to expose the plugin selection interface to the user).

6.3.2 Dynamic Analysis Framework

When performing parameter inference through dynamic analysis, we use a gray-box approach where the only input needed from the customer is their VM image. This allows us to maintain an as-a-service approach that does not modify the customer's underlying VM. While there are no specific input parameters, each DAF plugin does use a set of assumptions based on OS constructs (e.g., that an x86 OS uses privilege ring 3 for user applications and 0 for the OS kernel).

The only general assumptions made during dynamic analysis are that: (1) the cloud provider has access to the user's VM image and (2) the dynamic analysis framework can boot that VM image.

A1. The cloud provider has access to the VM's image

A2. The cloud provider can boot that image in an emulator

We can see that those assumptions hold true for both public and private clouds as the provider needs to have the user's VM image in order to run the user's VMs (**A1**). Many hypervisors use an emulator to provide virtual devices to VMs and therefore the VM image is compatible with an emulator (**A2**).

For the prototype DAF used in this work, we chose the open-source Dynamic Executable Code Analysis Framework (DECAF) [60] as introduced in Section 3.6.3. We chose DECAF because it is based on QEMU [9] and

therefore supports a VM image format that is compatible with the popular Xen and KVM hypervisors (satisfying **A2** from above).[2]

6.3.3 VM Monitoring Framework

For the VMF, we use hprobes, as introduced in Chapter 4. Using hprobes, monitoring plugins can be added and removed at runtime without disrupting the guest OS or hypervisor. In addition to hook-based monitoring functionality, we also add a set of callbacks to the KVM hypervisor to receive information about certain events of interest (e.g., on a VM Exit, an EPT page fault). In addition to the hprobe framework code, all of the monitoring API and RSaaS monitors presented later in this thesis are implemented as external kernel modules totaling 1680 lines of C code (including redundant boilerplate code used across different monitors). To help ensure that our modifications do not affect the robustness of the hypervisor, we confirmed that the KVM unit tests[3] pass on our modified version of KVM. It is important to note that the interfaces we add should not increase the attack surface of the hypervisor, as we do not add new transition points to and from the hypervisor and any data read from the VM is bounded and not used directly by the hypervisor.

6.3.4 Machine Configuration

Unless otherwise noted, all performance benchmarks for RSaaS detectors in Chapters 7-9 were performed on a machine with an Intel® Core™ i7-4790K CPU. The CPU has a clock frequency of 4.00GHz, the machine has 32GiB of DDR3 1333 MHz of RAM with a Hitachi HUA723020ALA640 7200RPM 6.0Gb/s SATA hard disk drive. This machine ran Ubuntu 14.04 LTS, though development also occurred on machines with various hardware running CENTOS7 and Ubuntu 12.04 LTS.

[2]Note that to support detailed analysis, DECAF uses QEMU in full emulation, whereas QEMU+KVM is used to run the VM at runtime

[3]http://www.linux-kvm.org/page/KVM-unit-tests

6.3.5 Prototype Discussion

Our main target for testing is the Linux OS (various distributions). Despite Linux being open-source, the cloud provider cannot use a white-box approach for inferring monitoring parameters since each distribution, or even each user, can configure the OS differently. We maintain our gray-box approach and only use OS semantics that can be obtained from our dynamic analysis or from version agnostic OS constructs (e.g., paging, published ABI, and privilege levels) and do not rely on or use the source code. Linux is a natural choice as a target OS for monitoring as-a-service as data from Amazon's EC2 shows that there are an order of magnitude more Linux servers running in EC2 (the most popular public cloud by usage statistics) than the next most popular OS [96]. Nevertheless, to demonstrate the versatility of our technique, we also present a keylogger detection example using Windows 7 in Chapter 9.

We can evaluate this prototype system partially in terms of the cloud computing aspects given in the NIST definition of cloud computing, as described in Section 2.6:

- **On-demand self-service:** This system is appropriate as the back-end for an on-demand monitoring service as it operates with the existing VM image as input and monitors can be added/removed at runtime.
- **Broad network access:** In addition to already being present in the IaaS system that RSaaS is built on top of, the assumption of broad network access is necessary for the deployment and transfer of VM images.
- **Resource pooling:** Once developed, monitors can be shared among multiple customers. Dynamic analysis only needs to be performed once per customer's OS kernel.
- **Rapid elasticity:** RSaaS monitors are elastic by their on-demand nature. Monitors can be added/removed and enabled/disabled at runtime without disrupting a running VM. In all experiments, monitors were added after the VM had been started.
- **Measured service:** The provider can measure differing levels of service based on the type and amount of monitors the user enables. Since

monitoring is controlled completely at the hypervisor level, it would be straightforward for the provider to accurately bill a cloud customer.

6.4 Discussion

The monitoring system presented in this dissertation is a hook-based system. Therefore, the aspect of the semantic gap that applies most directly to this work is building a high-level understanding of the guest OS's execution. Specifically, when the VM executes an instruction at an address, the hypervisor does not have an understanding of what higher-level operation is represented by that instruction. For hook-based monitoring, we solve the inverse problem: based on a monitoring specification, the cloud provider knows what operation they would like to monitor, but needs the address of that operation in order to add a hook.

6.4.1 System Model

In order to reason about our technique for bypassing the semantic gap, we consider an abstract model of an OS kernel. Since specific hardware architectures and OS implementations change, viewing the system from a modeling perspective helps us reason about how this work can apply to future systems. Figure 6.4 illustrates a classical model of an OS kernel from Iain Craig's *Formal Models of Operating System Kernels* [97]. In the OS model presented in Fig. 6.4, the main purpose of the OS is to run a set of user applications as processes that share the same hardware. These processes are isolated except for Inter-Process Communication (IPC) services provided by the OS. The processes are allowed to request assistance from the OS through a set of standard System Calls. Services provided in the lower levels of the kernel such as the Process Table and Low-Level Scheduler manage the processes.

In our approach, we identify semantic information about the guest OS that can be observed at the hardware layer. This semantic information is lower-level information (e.g., instruction addresses) that can be used to identify a higher-level OS construct (e.g., a context switch occurred). The term "OS constructs" used throughout this dissertation maps to the term "kernel primitives" in the classical kernel model. Since kernel primitives or OS constructs

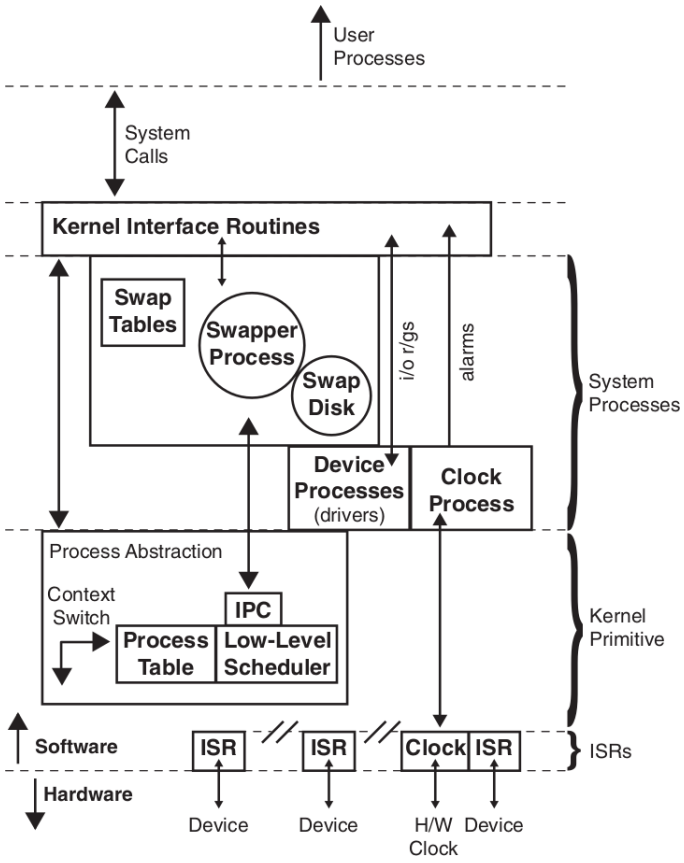


Figure 6.4: A classical model of an OS from *Formal Models of Operating System Kernels*. The work presented in this chapter focuses on using OS constructs (called kernel primitives in the model) to address the semantic gap in VM monitoring. Copyright 2007 Springer, used with permission under license number 3932820616060.

sit at the hardware/software interface, they can be inferred using hardware events (e.g., the low-level scheduler changes page tables). However, since these primitives are at the lowest software layer, they are version-agnostic (e.g., in Linux each process has its own virtual address space). We note that in many cases, one is interested in information in abstraction layers above kernel primitives (e.g., at the system call layer). When considering these higher layers, one must identify additional constructs for each abstraction layer. In the system call example, depending on the system call of interest, one would need to also consider the kernel interface routines and how those interact with the primitives before moving up to the system call layer. For example, we use information about the `init` System Process in Fig. 6.2 to identify the process creation system call. That said, including constructs from higher abstraction layers constrains the approach to a level of specificity that only applies to a single class or even version of an operating system (e.g., monitors based on POSIX conventions are not guaranteed to work for Windows).

Architectural Dependence

In Chapters 7-9, we present a set of example monitors implemented using the x86 architecture. An important question is how architecture dependent are the OS constructs we used to bypass the semantic gap for those monitors. For example, in the OS hang detector from Chapter 7, we use x86's `CR3` register to identify a process at the hypervisor level. The concept behind our choice for `CR3` is not restricted to x86. For example, the Translation Table Base Control Register is used to point to the current page tables in the ARM architecture. Even though we can find isomorphisms between x86 and ARM, our use of particular hardware events does restrict the individual detectors to processors that implement certain features. For example, in the OS hang detector, we assume that an MMU is present and paging is enabled, allowing us to use a particular register to identify processes.[4] Given this discussion, we believe that assuming an MMU is a reasonable assumption for a system that runs multiple processes. Even if in some future OS on a future architecture virtual memory is no longer used to isolate processes, one can expect another

[4] As a side note, if one were to use the hardware context switching features of x86, the Task Register `TR` could be used to uniquely identify processes, so paging is not strictly required.

hardware construct that would be correlated with processes running on the system.

6.4.2 Security and Reliability Models

The goal of this work is to improve the state of the art in reliability and security monitoring. However, the monitoring system presented cannot be used to detect all failures and attacks as it can be argued that it is impossible to prove that any one monitoring system detects all failures and attacks. Given this fundamental limitation of monitoring, it is important to understand what classes of failures and attacks can be detected by the system presented in this dissertation.

A hook-based VM monitoring system invokes the hypervisor when the VM reaches a certain point in its control flow. Therefore, most of the detection offered by this system involves detecting control flow related failures and attacks. Control flow based detection can be used to detect crash failures as the absence of progress on a certain control flow path is indication of incorrect behavior. Attacks that violate expected control flow can be detected based on an observed change in the execution of critical paths in the control flow (e.g., a function is or is not executed). If desired, one could build a complete control flow integrity (CFI) monitor using our system. This monitor would use the DAF to construct a control flow graph (CFG) and then ensure that the system does not deviate from its expected CFG [98]. However, VM Exit based CFI is not practical for production systems due to an extremely high expected overhead from inducing a VM Exit at every branch instruction. For data-based attacks, we can detect attacks that have a measurable influence on the control flow. For example, the keylogger detector in Chapter 9 detects an attack when keystroke data flows to a malicious process by observing that the malicious process is scheduled after keystrokes.

6.4.3 Interface

The intended use of RSaaS is for the cloud provider to develop trusted monitoring plugins that are then shared with a cloud customer through the standard cloud interface. Based on the current prototype that uses kernel mod-

ules, we do not expect providers to open the ability to develop monitoring plugins to cloud users without additional features to protect this hypervisor-level development from affecting other users' VMs. The skill required to develop DAF (DECAF) and VMF (hprobe) plugins is roughly the same as that required for kernel module development, and this effort can be amortized by reusing the plugins across customer VMs. Since cloud providers run extremely large systems and have administrators with expert-level OS experience, we do not view the skill requirement as detrimental to the adoption of our technique.

Chapter 7

OS Hang Detection

7.1 Motivation

One of the biggest limitations of cloud computing is the lack of physical access to computing resources. Since users must access their cloud resources through a network interface (e.g., VMs), a lack of responsiveness can be due to either a network failure or a system failure. To help isolate the possibility of network failures and diagnose system failures due to issues with the guest OS, we introduce an OS hang detector. Like the "smart" detectors in Chapter 5, this hang detector demonstrates the concept of dynamic monitoring to increase the performance of a monitor.

While some hypervisors provide a watchdog device,[1] that device requires a guest OS driver and therefore is not amenable to an as-a-service approach. We propose a hang detector based on the activity of the guest OS scheduler. Our approach requires no drivers or configuration from the guest OS and can be added at runtime. Furthermore, though OS hangs are not the most common failure mode, they are catastrophic. However, as any organization running large scale systems understands, even the most unlikely failure modes become common at scale and IaaS clouds represent some of the largest computing systems in existence [99].

The main concept behind the OS hang detector is that regardless of whether processes are running or not, the OS scheduler runs periodically. This detector works by reporting a hang if the scheduler does not execute when it is expected to.

[1] https://libvirt.org/formatdomain.html#elementsWatchdog

7.2 Hang Detector DAF Plugin

A hook-based hang detector based on how often the scheduler runs requires two version-specific guest OS parameters: (1) the address of the scheduler and (2) the maximum expected interval between invocations of the scheduler. A guest OS hang is therefore detected when the instruction at the scheduler's address is not executed within the maximum expected interval, similar to a traditional heartbeat detector (as presented in Section 5.1). Measuring the maximum scheduling interval allows us to choose a timeout value that minimizes detection latency but also does not have false positives. The parameter inference step of the OS hang detector locates the address of the scheduler and then profiles the scheduler to determine a timeout interval for the detector.

Note that in earlier versions of the Linux kernel, the scheduler was invoked at a regular interval based on a timer "tick." The tick was configurable and defaulted to 1000Hz (one tick is called a *jiffy*, a measure of time used throughout kernel code). Recent kernels, however, have moved to a "tickless" approach to reduce power consumption [100,101]. With a tickless kernel, the scheduler no longer runs at a fixed frequency, so the maximum measured scheduler interval depends on OS configuration and the software installed (i.e., systems running a variety of applications will have more scheduler invocations). We emphasize that while the there is no well defined timer tick in tickless kernels, there are other activities that occur while the system is idle and an idle system should still have an upper bound on the maximum time between scheduler invocations.

In order to identify the address of the scheduler, we recognize that the scheduler will update the `CR3` value when changing virtual address spaces as part of a process switch. While other functions also write to `CR3`, we have observed that the scheduler is the only function that consistently writes to `CR3` over time. This leads to a simple heuristic: the scheduler is simply the function that writes to CR3 the most. However, the instruction of the scheduler that writes the `CR3` register is not called on every invocation of the scheduler: it is only called when processes are changed. To identify the scheduler, which is the calling function from the address of the instruction that performs the `CR3` write, we read the VM's memory backwards from the location of that instruction. We read backwards until we identify a function header

```
1: procedure ON_INSTRUCTION_END(cpu_state)
2:     if sched_address != NULL then
3:         delta_t ← last_time - current_time
4:         if delta_t > largest_delta_t then
5:             largest_delta_t ← delta_t
6:         end if
7:         last_time ← current_time
8:     else
9:         if instruction == mov *, cr3 then
10:            if last_eip == current_eip then
11:                if  count ≥ threshold then
12:                    sched_address ← function(eip)
13:                else
14:                    count+=1
15:                end if
16:            else
17:                last_eip ← current_eip
18:                count ← 0
19:            end if
20:        end if
21:    end if
22: end procedure
23: procedure FUNCTION(address)
24:    /*Find the calling function that contains this address*/
25:    function_not_found ← false
26:    while function_not_found do
27:        if *address == push %ebp;mov %esp,%ebp then
28:            return address
29:        end if
30:        address-=1
31:    end while
32: end procedure
```

Figure 7.1: Pseudocode for OS hang detection. This procedure is executed after every instruction in DECAF. In x86, `EIP` is the register containing the instruction pointer. Safety measures in the actual code (e.g., a maximum search threshold) are omitted for brevity.

(using the gcc calling convention for x86, the assembly function call header is `push %ebp;mov %esp, %ebp`). To identify the maximum scheduling interval, we boot the VM image and monitor calls to the identified scheduler. Pseudocode for the component implemented in DECAF is presented in Fig 7.1. In this implementation, we decided to use the instruction end callback (DECAF_INSN_END) as we wanted to showcase the use of calling conventions to obtain the start of a function. Alternatively, we could have used a different callback (e.g., `DECAF_BLOCK_END_CB` or `DECAF_OPCODE_RANGE_CB`) to track `call` instructions leading up to the `CR3` write. While we did not encounter Linux with in-kernel Address Space Layout Randomization (ASLR) [102,103] in our experiments, if the system is using in-kernel ASLR, an offset from a fixed location in the kernel text section (e.g., from the `SYSENTER_EIP` MSR) as opposed to the scheduler's address could be used since both the scheduler address and system call handler are in the text section of the main kernel.

The hang detection analysis plugin makes the following assumptions:

1. A process change can be signaled by a change of virtual address spaces (write to `CR3`).

2. `CR3` writes are performed by a function that uses an expected calling convention (we can also use alternate heuristics, such as tracking `call` instructions).

3. The system boots into an idle loop or a state that would exhibit a minimal scheduling frequency (e.g., the user's application with no workload).

We find the interval measured from the DAF plugin to be an upper bound on the scheduling interval. The OS and any applications configured to start at boot will enter an idle state after boot. At run time, we expect more activity due to input and therefore more scheduling events. Furthermore, we introduce overhead from emulation and dynamic analysis, which slightly inflates the measured scheduling interval.

Table 7.1 summarizes the results from running the DAF plugin for finding the scheduler address and maximum measured scheduling interval on various kernel versions. Note that for Fedora 11 the plugin did not identify the scheduler. However, the hang detector will still detect a kernel hang for Fedora 11 as the `switch_mm` function is called when processes are changed (process

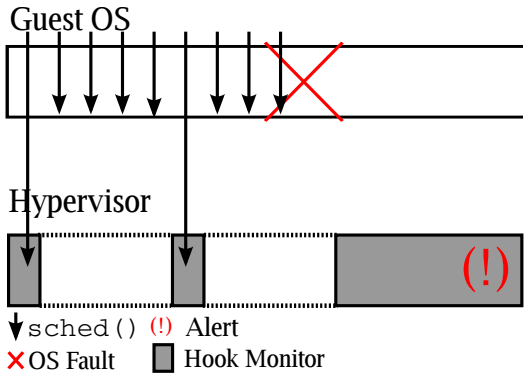


Figure 7.2: Dynamic monitoring example: the hypervisor is notified when the scheduler runs. During a hang the hook is still added but the scheduler does not run.

changes have been used for hang detection in the past [8]). Even though hangs will still be detected by watching `switch_mm` instead of `schedule`, the frequency at which `switch_mm` is called is lower than `schedule`, causing a higher detection latency when using `switch_mm`.

7.3 Hang Detector VMF Plugin

The runtime monitoring plugin works by adding an hprobe to the identified scheduler address. If the time between hprobe invocations exceeds the maximum expected scheduling interval, the system raises an alert. There is a performance concern with adding hprobes to the scheduler as hprobes work by using hardware-enforced VM Exit events. If one generates a VM Exit event on every scheduler invocation, there could be significant overhead (e.g., for the traditional 1000Hz timer there would be 1000 VM Exits per second). However, to detect an OS hang we do not need to hook every single call to the scheduler. Instead, we can take a dynamic monitoring approach. We add an hprobe to the scheduler and when that hprobe is activated, we remove the hprobe. We then queue another hprobe to be added at the scheduler's address after the expected scheduling interval. This dynamic monitoring approach is illustrated in a timing diagram presented in Fig. 7.2.

Table 7.1: Functions Identified as the Scheduler by Dynamic Analysis

OS	Function Name	Interval (s)	Kernel Version
CENTOS 5.0	`schedule`	3.5193	2.6.18-8.el5
CENTOS 5.4	`schedule`	0.2507	2.6.18-398.el5PAE
Fedora 11	`switch_mm`	20.0120	2.6.29.4-167.fc11.i686.PAE
Ubuntu 10.10	`schedule`	1.0077	2.6.35-32-generic-pae
Arch Linux	`__schedule`	1.7563	3.17.6-ARCH

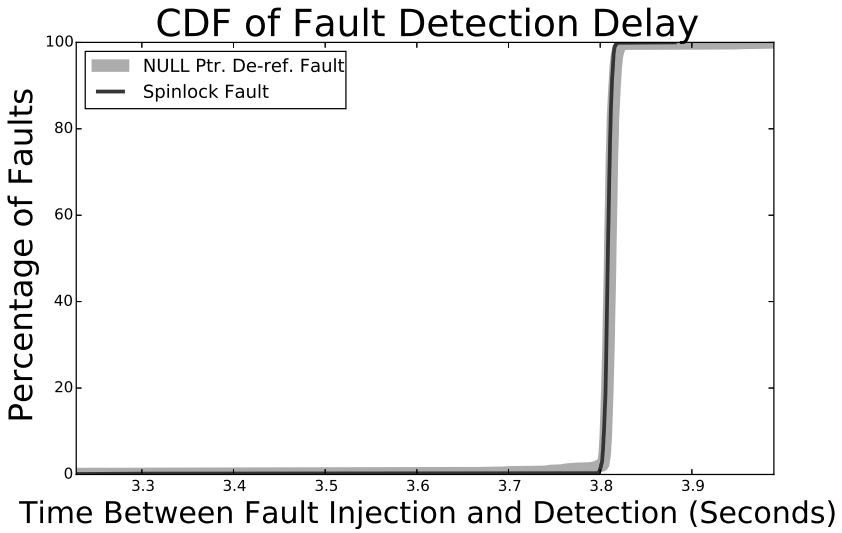


Figure 7.3: The CDF of detection latency of a system hang for both a deadlock and NULL pointer de-reference.

7.4 Hang Detector Evaluation

In order to evaluate the functionality of the hang detector, we performed fault injections using double spinlocks and NULL pointer dereferences to hang the kernel. For these experiments, we used the Ubuntu 10.10 guest. To measure the detection latency (the time from when a fault is injected to when that fault is detected) and ensure the robustness of our detector against race conditions, we repeated both the double spinlock and NULL pointer dereference injections 1000 times each. For both fault types tested, the detection coverage was 100% with 0 false positives and 0 false negatives. The cumulative probability distribution for the detection latency is plotted in Fig. 7.3.

We evaluated the performance benefits of dynamic monitoring with a con-

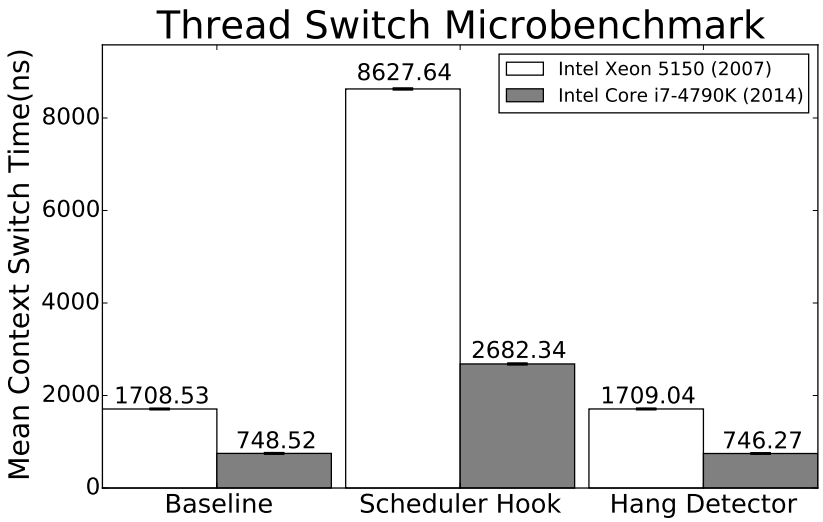


Figure 7.4: Context switch microbenchmark. The baseline is a VM without the hang detection monitor. The scheduler hook represents a naïve approach where a hook was always added to the scheduler. The last data set represents the dynamic monitoring approach.

text switch microbenchmark.[2] The benchmark measures the time the OS takes to switch between two threads. Switching two threads will invoke the scheduler but almost nothing else (i.e., no tasks that access devices or change `CR3`). We ran the benchmark 30 times on a VM without any hooks, with hooks always added to the scheduler (the naïve approach), and with the dynamic monitoring approach. From Fig. 7.4, we can see that the overhead of the naïve approach is significant, but by using our dynamic approach the resulting overhead, even in a microbenchmark, is negligible. For comparison, the experiments that executed hprobes for every scheduling event involved 3,002,135 probe invocations compared to only 50 for the dynamic monitoring experiments. We also note that as was observed in previous experiments (see Chapter 5), the newer generation hardware exhibits much lower VM Exit overhead.

To gauge the performance impact of this detector on cloud applications, we run three application benchmarks: a compile of Linux Kernel 2.6.35, Apache Bench, and PostMark. Apache Bench and PostMark were both configured and run through Phoronix Test Suite and all three were run 30 times.[3] Apache Bench is used to represent a traditional webserver workload and is

[2] https://github.com/tsuna/contextswitch
[3] http://www.phoronix-test-suite.com/

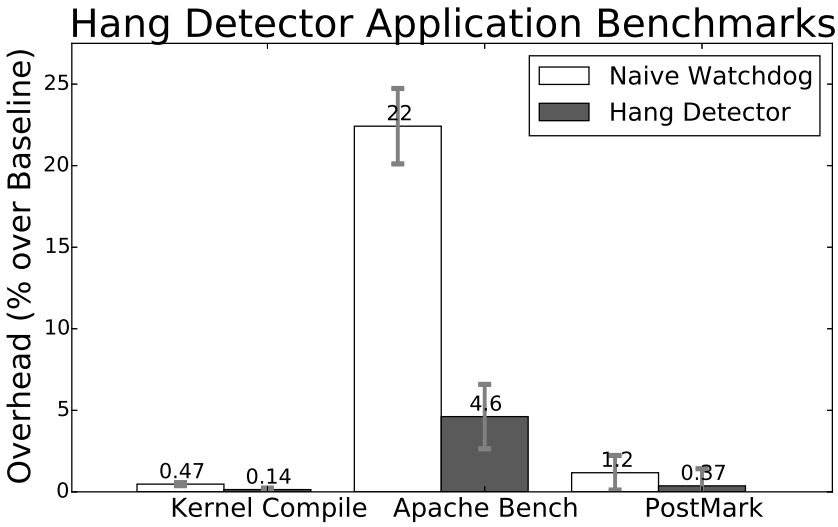


Figure 7.5: Application overhead comparing hang detection methods. The naïve approach hooks every schedule call. The final column is the dynamic monitoring. Lower is better for all benchmarks.

evaluated in terms of requests per second. PostMark is used to measure disk performance and is evaluated in terms of transactions per second. All of these experiments were performed on an Ubuntu 10.10 VM. Fig. 7.5 shows the results of this evaluation with error bards indicating the 95% confidence interval of the mean.

All benchmarks were run with no monitor loaded, a naïve monitor, and with our dynamic hang detector. In Fig. 7.5 the impact of our hang detector over the baseline is negligible in all three cases, reducing the mean performance by 0.38%, 4.41%, and 0.34% for the kernel compile, Apache Bench, and PostMark, respectively.

Chapter 8

Return to User Attack Protection

8.1 Motivation

Return-to-user (ret2user) attacks are attacks where userspace code is executed from a kernel context. An illustration of a ret2user attack is shown in Fig. 8.1. Ret2user is a common mechanism by which kernel vulnerabilities are exploited to escalate privileges, often using a NULL pointer dereference or by overwriting the target of an indirect function call [104]. Ret2user is simpler for attackers than using pure-kernel techniques like Return Oriented Programming (ROP) since the attacker has full knowledge and control over their shellcode in user space, and only needs to trick the kernel into executing that shellcode (as opposed to deriving kernel addresses or figuring out a way to copy shellcode into kernel memory). Preventing the ret2user class of attacks raises the effort needed to exploit a kernel vulnerability. If a ret2user vulnerability cannot be used to escalate privileges, it can be used to crash a system via a Denial of Service (DoS) attack by causing a kernel-mode exception. In most cases attackers would prefer the privilege escalation outcome, but users are still harmed if an exploit attempt results in a crash/DoS. We use the ret2user attack as an example of how to build a security detector for the RSaaS framework that is based on OS constructs that apply to multiple vulnerabilities.

8.2 Detector Design

A ret2user attack can be detected by checking if code in a user page is executed from a kernel context. In Linux, the kernel's pages are mapped into every process's address space at the same virtual addresses to reduce TLB

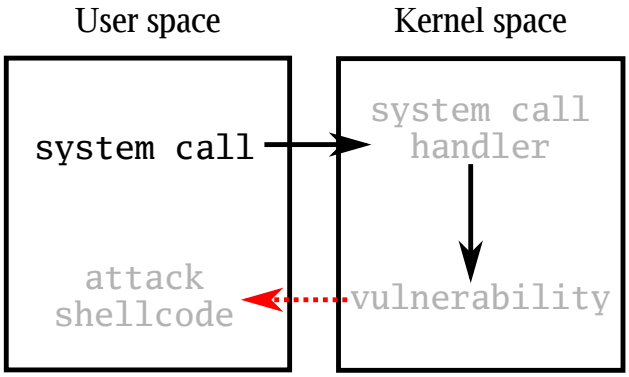


Figure 8.1: A return to user attack. All code executed with system privileges is gray. In this example, a vulnerability in a system call handler causes kernel control flow to execute code in userspace with system privileges. The dashed arrow is the execution of user code with kernel privileges that is prevented by the detector presented in this chapter.

pressure for user/kernel context switches. While during certain operations (e.g., disk I/O) the kernel is expected to copy data to/from user-level pages in memory, the kernel is never expected to execute code inside user pages. The ret2user detector works by forcing user pages to be non-executable while the CPU is executing in a kernel context using EPT permissions. Trying to execute userspace code will result in a VMExit back to the hypervisor.

In order to detect when userspace code is executed from a kernel context, the ret2user detector must know when the guest is executing in kernel mode (ring 0) and when it is executing in user mode (ring 3). Unfortunately, changes to the Current Privilege Level (`CPL`) do not trap to the hypervisor. Our ret2user detector overcomes this issue by identifying the valid entry and exit points to and from the kernel. Hprobes are then added to those points so the hypervisor can track whether the guest is executing in kernel or user mode.

8.2.1 Return-to-User DAF Plugin

The parameters for the ret2user detector are the entry and exit points to and from the kernel. In particular, we must be sure to infer all valid entry and exit points in the kernel. This includes identifying the entry and exit points that could be used during a ret2user attack. The OS constructs we use for inferring the entry and exit points to and from the kernel are:

```
1: procedure ON_INSTRUCTION_END(cpu_state)
2:     if last_cpl == 3 && cpu_state.CS.sel == 0 then
3:         /*Transition from user to kernel*/
4:         KERNEL_ENTRIES ∪ cpu_state.EIP
5:     else if last_cpl == 0 && cpu_state.CS.sel == 3 then
6:         /*Transition from kernel to user*/
7:         /*The EIP of the previous instruction is a kernel address*/
8:         KERNEL_EXITS ∪ last_eip
9:     end if
10:    last_cpl ← cpu_state.CS.sel
11:    last_eip ← cpu_state.EIP
12: end procedure
```

Figure 8.2: Identifying kernel entry and exit points. The processor's current privilege level (`CPL`) is stored in the selector of the `CS` segment register.

1. The kernel runs in ring 0 and the user applications run in ring 3.
2. The kernel entry/exit points are finite and will not change across. reboots

In the DAF plugin for the ret2user detector, we track the `CPL` after each instruction was executed and record the instruction pointer (the `EIP` register) when the CPL transitions from 0→3 or 3→0. Note that since we want to track the entry and exit points in the kernel (as the userspace transition points are application dependent and non-deterministic), we use the address of the previous instruction (a kernel address) for the 0→3 CPL transition as opposed to the current instruction in `EIP` (a user address). The pseudocode for the DAF plugin is shown in Fig. 8.2.

8.2.2 Return-to-User VMF Plugin

The runtime monitor for the ret2user detector adds hprobes to the kernel entry and exit points obtained during the parameter inference step. After the VM boots, we scan the guest page tables to identify which guest virtual pages belong to the kernel (those page table entries have the User/Supervisor bit cleared). Note that this process for scanning which pages belong to the kernel could not be performed during the dynamic analysis step in the event that kernel modules are loaded to different addresses or that the guest OS

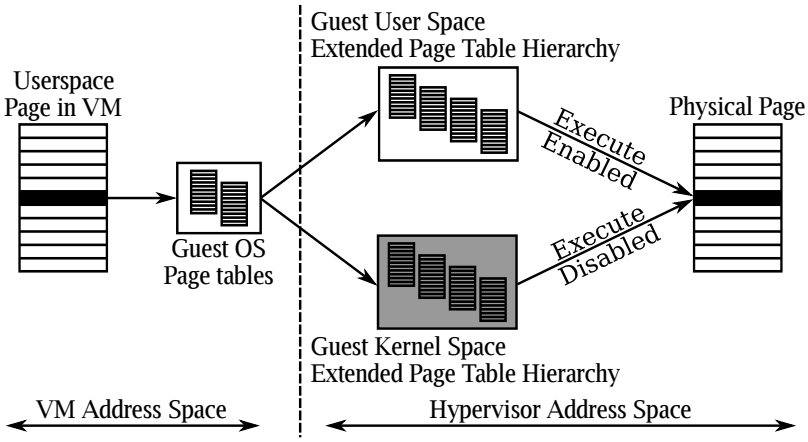


Figure 8.3: Ret2user attack detector. When the VM transitions from guest user to guest kernel space, the in-VM hooks notify the hypervisor and the hypervisor switches EPT address spaces. In the guest kernel address space, EPT entries for guest user pages have the execute bit cleared to prevent ret2user attacks. The VM controls its own page tables, but is isolated from editing the EPTs.

uses Address Space Layout Randomization (ASLR) (we can compensate for the changes in entry/exit points by using an offset from the `SYSENTER_EIP` MSR that contains the system call entry point, as discussed for OS hang detection in Section 7.2).

After obtaining the virtual addresses for the kernel's code pages, we create a second set of EPTs in addition to the tables created by the hypervisor when the guest was started. We then copy the last-level EPTEs to the new tables so that the last-level entries still correctly point to the host pages containing the VM's data. When copying the last-level entries, we remove execute permissions. We then add hooks to the VM at the obtained entry/exit points for the kernel. We switch the set of active EPT tables at each transition: we use the original tables while the guest is executing in user mode and the second set of tables while the guest is executing in kernel mode. Fig. 8.3 illustrates the ret2user detector.

After removing execute permissions from the kernel's pages in the second set of EPT entries, an `EPT_VIOLATION` VM Exit event is generated whenever the guest executes code in guest kernel pages. We added a callback to KVM's `handle_ept_violation` function that handles those EPT violations. Inside the callback we check the guest virtual address at which the EPT violation occurred. If the EPT violation was generated by an attempt at executing

code in an address belonging to a user page, we know that a ret2user attack was attempted. If the EPT violation resulted from execution in a kernel page, then we remove execution protection for that page. In this way, the kernel context is lazily built in the EPT structures. We note that this detection scheme does not protect against new transitions from guest user space to kernel space (i.e., those added by attackers and not discovered by the DAF plugin). However, if the attack code returns to userspace, we could detect that the kernel exit did not have a corresponding kernel entry. That said, all the vulnerabilities we studied used the system call entry and exit points. If the EPT violation was due not to an execute attempt but to an unrelated read or write EPT violation (for example, if it was just the kernel loading a page to a guest physical address that had not yet been used), then we remove execute permissions from the page if it is a user page. This allows us to maintain the behavior of the ret2user detector as new applications are started. Note that in addition to editing permissions to detect vulnerabilities, we synchronize the EPT tables across updates and also invalidate any entries as they are updated (to avoid issues similar to having stale TLB entries).

8.3 Return-to-User Evaluation

The ability of the ret2user detector to detect attacks hinges on correctly determining whether or not the VM is in user or kernel space. When inferring parameters for the runtime detector, we boot the VM while running dynamic analysis. Since we are performing dynamic profiling, it is important to understand how workload dependent the analysis is. To test the coverage of observed kernel entry/exit points, we profiled a test VM running CENTOS 5.0 (we chose this guest OS because it contained multiple vulnerabilities). First, we collected the entry/exit points from a bootup and shutdown sequence. Note that we can send an ACPI shutdown command via `system_powerdown` in QEMU to shut down the VM without user input. To test whether a bootup/shutdown sequence is sufficient, we also measured the entry/exit points with a Linux kernel source archive download, extraction, and compilation. This download, extract, compile workload exercises the kernel entry/exit points one would expect to see during a VM's lifetime: downloading a file exercises the network and disk, while extracting and com-

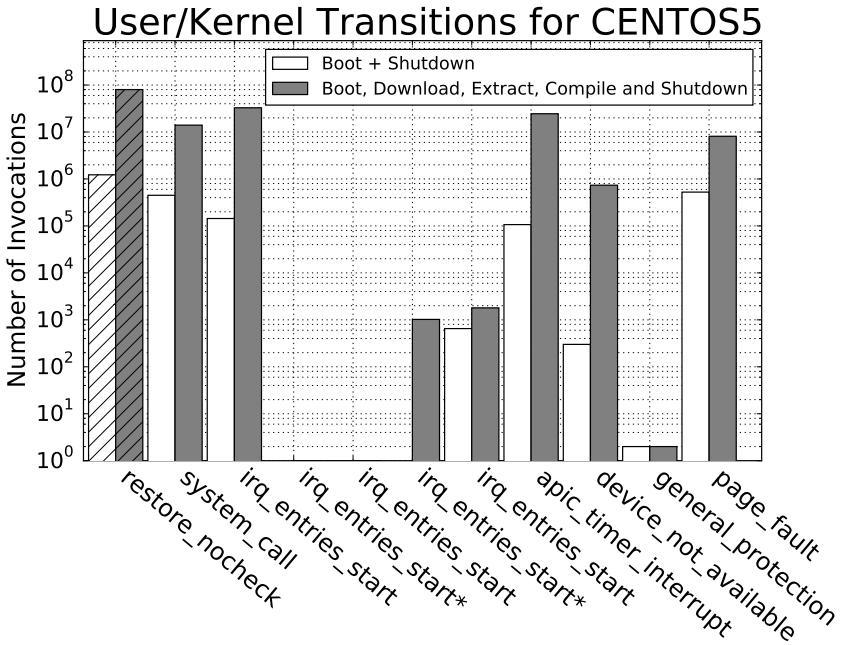


Figure 8.4: Parameter inference for the ret2user detector. The vertical axis indicates the number of times those points were invoked and the horizontal axis indicates the function containing those points. The diagonal hatched/solid bars represent guest kernel exit/entry points, respectively. `irq_entries_start` appears multiple times as each IRQ line represents a unique kernel entry point ($*$ denotes the transitions that only appeared with the test workload).

Table 8.1: Ret2user Vulnerabilities Detected

OS	Vulnerability	# Entries	# Exits
CENTOS 5.0	CVE-2008-0600	7	1
CENTOS 5.0	CVE-2009-2692	7	1
CENTOS 5.0	CVE-2009-3547	7	1
Fedora 11	CVE-2009-2692	7	1
Ubuntu 10.10	CVE-2010-4258	6	1

piling are mixed cpu/disk/memory workloads involving the scheduling of multiple processes.

The results of the kernel entry/exit point profiling experiments are shown in Fig. 8.4. The only entry points that were observed during the kernel workload and not during bootup/shutdown were entries used in IRQ handling (those entries were activated 9 and 3 times). If needed, one could supplement the entry points inferred by looking up interrupt handler entries in the interrupt tables. All ret2user exploits we studied use the system call entry point, including exploits involving vulnerabilities in the kernel's interrupt handling code.[1] To measure the effectiveness of the ret2user detector, we tested it against public vulnerabilities as shown in Table 8.1. We observe that in the kernels tested, we only identified one common exit point into userspace. That said, by definition the ret2user detector cannot protect against exploits that stay in kernelspace (e.g., those using pure-kernel ROP). We currently do not implement any response techniques once an attack is detected. The simplest mitigation is just to never restore execute permissions to those pages and prevent the exploit from ever executing. However, that simple mitigation technique may disrupt the OS's execution if the kernel is non-preemptible as the kernel thread handling the ret2user path may either hang or triple fault (if the page fault gets forwarded to the VM).

The ret2user detector cannot be circumvented by a guest unless a user in the VM compromises the hypervisor or creates a new kernel entry/exit point. The ret2user detection technique is general and can detect ret2user exploits using yet-to-be-discovered vulnerabilities. To highlight the significance of ret2user attacks, Intel has released a similar protection in hardware called Supervisor Mode Execution Protection (SMEP) or OS Guard (see Section 4.6 of the Intel Software Developer's Manual [16]). SMEP effectively offers

[1] https://www.exploit-db.com/exploits/36266/

protection similar to our detector, but since SMEP is configured by the guest OS it (1) requires support in the guest OS, and (2) can be disabled by a vulnerability in the guest OS [105, 106]. Unlike SMEP, the ret2user detector can also be used to protect VMs which are running legacy OSs (a common virtualization use case) or on CPUs that do not support SMEP. This detector is more flexible than SMEP, and if one wishes, one could change the criteria for what is protected beyond pages that have the User/Supervisor bit (e.g., to restrict code from unapproved drivers or system calls [107]).

The ret2user detector triggers a VM Exit on every transition between guest user and guest kernel space. To measure the overhead of the detector, we ran a kernel uncompress and compile as well as a disk write and kernel entry/exit microbenchmark. The disk write benchmark copies 256 MiB from `/dev/zero` to `/tmp` (the buffer cache is cleared on every iteration) and the microbenchmark is the same except it writes to `/dev/null` to remove any disk latency and effectively exercises only kernel entry/exit paths.

The results of the performance measurements for ret2user are given in Fig. 8.5. The microbenchmark exhibits roughly 20x overhead, but the kernel workloads offer 0.15x overhead. Additionally, we reran the same filesystem and web workloads from Section 7.4. The results for Apache Bench and PostMark can be seen in Fig. 8.5. The ret2user detector adds 77.49% overhead for Apache Bench and 42.68% overhead for PostMark, respectively. Our technique's ability to change its monitoring functionality at runtime makes it an ideal platform for a future adaptive monitoring system [108]. The adaptive system could, for example, use more expensive security monitors (e.g., ret2user) only when lower overhead monitors detect suspicious activity (e.g., one sees `gcc` executed on a webserver) [67].

Vulnerabilities are common, but released infrequently on computing timescales. As of writing, 192 privilege escalation vulnerabilities have been identified in the Linux kernel since 1999, corresponding to roughly one new vulnerability being announced every month.[2] Even if an organization is using vulnerable software, it is unlikely that every vulnerability discovered for that software applies to the configuration used by that organization. However, clouds by their nature are heterogeneous (many customers running various applications and configurations). Therefore, a provider can reasonably expect that any

[2] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

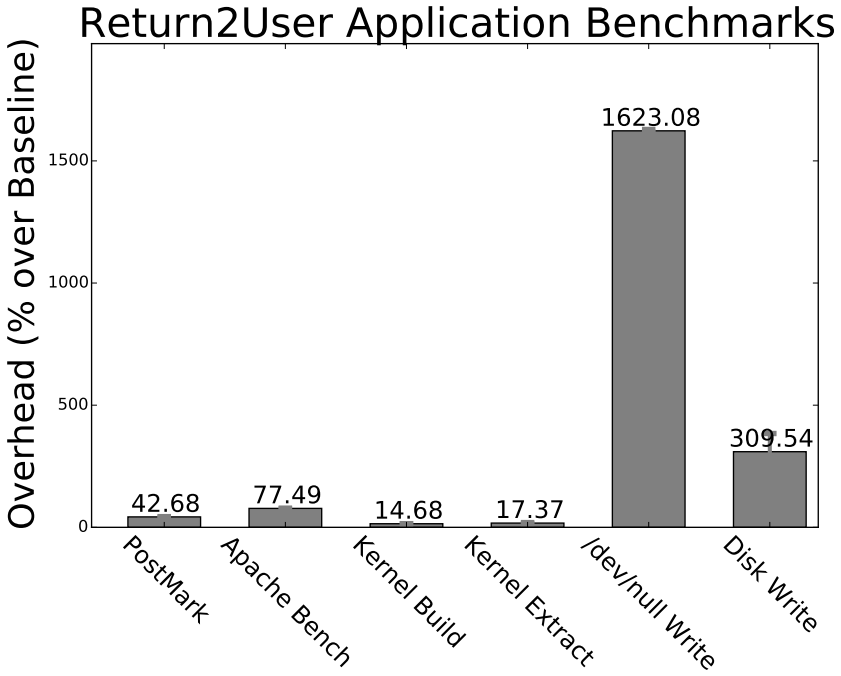


Figure 8.5: Benchmark overhead for Apache Bench, PostMark, a kernel source extract, build, and microbenchmarks focused on writes compared against a baseline. The baseline is run without the ret2user monitor. Lower is better.

given vulnerability will apply to a subset of that provider's users and enable a detector like ret2user to mitigate risk before systems can be patched. A performance cost during this period can be preferable to either running unpatched systems or disrupting a system for patching. However, some users may treat security as a top priority and protect against zero-day attacks by running the detector continuously. Customers can chose to place different detectors on their internet-facing and backend (e.g., database) servers. Since our method modifies no guest OS state (other than adding hprobes) and can be enabled at run time, VMs could also be live migrated to an analysis or monitoring environment. In a cloud environment, therefore, it becomes desirable to protect against high-impact vulnerabilities even if they are released relatively infrequently. Similar to the motivation behind the Emergency Exploit Detector in Chapter 5, vulnerabilities are not guaranteed to be released on a set schedule and can therefore be announced at any time: activating any mitigation technique should aim for minimal disruption of both the users' and provider's systems.

8.4 Related Work

The ret2user detection in this chapter can be viewed as a subset of SecVisor's detection techniques [45]. Instead of Intel EPT, SecVisor uses the functionally equivalent Nested Page Tables (NPT) from AMD to create isolated address spaces for kernel and userspace. However, instead of inferring entry and exit points, SecVisor modifies the guest kernel (requiring a recompile) by adding hypercalls to inform the hypervisor about guest OS operation. It should also be noted that SecVisor was implemented standalone and is similar to Intel's kernel guard [46]: SecVisor runs underneath a single guest OS and does not support multiple VMs. Therefore, SecVisor is not suitable for use in an IaaS cloud context (unless one uses performance costly nested virtualization techniques [11]).

SecPod [109] is used to protect the guest kernel's page tables. Like the VM Exit techniques discussed in Chapter 3, SecPod ensures guest kernel space is protected by reconfiguring the hypervisor to trap on privileged operations. SecPod uses a secure address space for auditing the guest kernel's paging operations. This secure address space is transitioned to and from using spe-

cial entry and exit gates that are added manually to the guest kernel. One could use the techniques for detecting kernel entry and exit presented in this chapter to implement SecPod-like functionality without modifying the guest kernel. Parameter inference would discover all the paging functions and runtime monitoring would hook on those functions. One would also enable the same VM Exits SecPod uses to protect itself.

Chapter 9

Process-based Keylogger Detection

9.1 Motivation

Many enterprise environments use Virtual Desktop Integration or VDI to provide workstations for their employees. The difference between a traditional desktop environment and a VDI environment is illustrated in Fig. 9.1. In VDI deployments, each user's desktop environment is hosted on a remote VM inside a datacenter or cloud. The user then connects to the VM image using either a software or hardware thin client. VDI offers many advantages including a simpler support model for (potentially global) IT staff and better mitigation against data loss (e.g., from unauthorized copying of data to external media). While VDI provides security benefits due to the isolation offered by virtualization, VDI environments are still vulnerable to many of the same software-based attacks as traditional desktop environments. One such attack is a software based keylogger that records all keystrokes inside a guest OS.

Process based keyloggers are keyloggers that run as processes inside the victim OS. These keyloggers represent a large threat as they are widely available and easy to install due to portability. Previous work in keylogger detection is built on looking at I/O activity as keyloggers will either send data to a remote host or store the keystroke data locally until it can be retrieved. In this section, we present a new detection method for process based keyloggers that monitors for changes in the behavior of the guest OS [110].

The main concept behind detecting process-based keyloggers is quite simple: after a keystroke is passed into the guest OS, the keylogger process will run to consume and record that keystroke. This keylogger process will run in addition to any other processes that usually consume keystrokes (e.g. for Windows, a system process like `csrss.exe` or the actual user application like

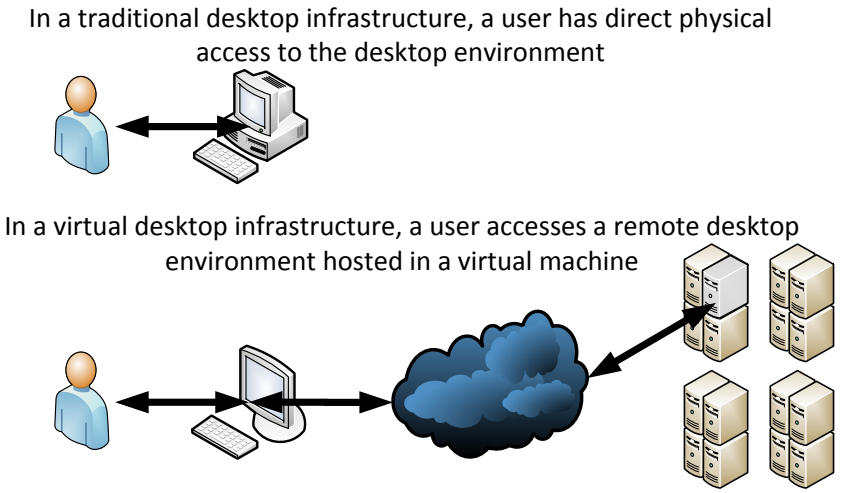


Figure 9.1: A comparison of a traditional enterprise desktop environment and a desktop environment using Virtual Desktop Integration.

`notepad.exe`).

9.2 Keylogger Detection Parameter Inference

The DAF plugin for keylogger detection observes how the guest OS responds to a hardware event from the keyboard. Just like physical keyboards connected a desktop computer, a virtual keyboard connected to a VM will generate an interrupt which will then be processed by an interrupt service routine (ISR). In x86, the ISRs are stored in the Interrupt Descriptor Table (IDT). The goal of the DAF plugin is to identify which ISR is responsible for handling keyboard interrupts as different VM instances may use different IDT entries or even different virtual devices.

In order to identify the keyboard interrupt handler, we added a callback to DECAF that tracks all hardware interrupts (`DECAF_INTR_CB` in Table 3.2). We boot the VM in DECAF and send keyboard input to the VM. Using the hardware interrupt callback, we determine the IDT entry for the keyboard interrupt handler as well as the EIP of the keyboard interrupt handler. Even though this analysis plugin uses keyboard input, we can send keyboard events through software and can still perform parameter inference automatically without user interaction. DECAF exports QEMU's `do_sendkey` function to plugins. This function sends keyboard input to the emulated target using the same internal codepaths as the normal virtual keyboard. The `do_sendkey`

function allows us to maintain the same aspect of the other detectors in only needing a system boot to obtain the keyboard interrupt handling parameters.

9.3 Keylogger Detection Runtime Detector

The detector takes as its input the IDT entry number. When the keylogger detector is enabled, a hook is then added to the ISR for the keyboard interrupt as determined during the dynamic analysis step. Whenever a key is pressed, the detector re-enables `CR3` VM Exits and the `CR3` values after the keystroke are recorded and analyzed as described below.

As mentioned in the previous section, our goal is to detect the presence of a keylogger based on changes in scheduling behavior. In order to do so, we devised a metric based on the number of processes responding to a keystroke. Each process-based keylogger uses slightly nuanced hooking methods (none that we tested overrode IDT entries), but all of the keyloggers we tested get scheduled shortly after a keystroke. This allows us to build a simple but effective detection heuristic: the more processes (represented by `CR3` values) responding to a keystroke, the more likely it is that a keylogger is present.

We call the per-process metric we used to measure a keylogger the *responsiveness score* for that process. The responsiveness score is computed for each process at every keystroke by exponential decay with a half-life of $t_{1/2}$ as shown in Eq. 9.1:

$$R_k(\texttt{CR3}) = e^{-\ln(2)\frac{t_{CR3}-t_k}{t_{1/2}}} \tag{9.1}$$

$R_k(\texttt{CR3})$ is summed over every CR3 change after keystroke k and before keystroke $k+1$. If that sum is ≥ 1 for any process, we count that process as responding to keystroke k. While detector is running, we measure the mean value of R_k across all processes and when it is greater than an expected threshold (the threshold discussed during the evaluation below), we report the presence of a keylogger.

In order to measure $R_k(\texttt{CR3})$ at runtime, we add a hook to the ISR responsible for keyboard events and re-enable `CR3` VM Exits. We compute compute $R_k(\texttt{CR3})$ in a separate analysis program running in userspace on the hypervisor.

9.4 Keylogger Detection Evaluation

In our experiments we use a half-life of $t_{1/2} = 100$ms. This was chosen based on the perception time of the average person [111]. Note that this keylogger detector is only an example detector for our RSaaS and not an exhaustive study on keylogger detection, so it is likely that there exist better parameters or functions for measuring responsiveness. We tested the parameter inference on Ubuntu 16.04, Windows 7, and Windows 8 and found that the IDT entries for the keyboard were 0x31, 0x91, and 0x90, respectively. For a more thorough evaluation we used a Windows 7 VM and tested the detector against four keyloggers freely available from the Internet. The keyloggers we tested were: Revelear Keylogger, Actual Keylogger, Spyrix Keylogger, and Free Keylogger Platinum. We tested each keylogger with multiple workloads: typing in notepad, browsing in Internet Explorer, browsing in Mozilla Firefox, editing a spreadsheet, editing a slideshow presentation, and running ten background processes while editing a document. We also ran the same workloads with no keylogger present and the Receiver Operating Characteristics of our experiments are plotted in Fig. 9.2. We see that this basic keylogger detector performs well, with a AUC of 0.95 (an ideal detector would have an AUC of 1.0). We found that the false negatives came from experiments with the Actual Keylogger in web browsing workloads. After inspecting our experimental data, the Actual Keylogger does not log keystrokes from Internet Explorer and therefore was not detected.

Note that due to machine availability, the keylogger detection performance benchmarks were conducted on a different platform than the other monitors. The CPU was a 14 core Intel® Xeon™CPU E5-2683 v3 @ 2.00GHz and the machine had 256 GB of LR-DIMM DDR4 memory operating at 2133 MHz. To measure the performance impact of the keylogger detector, we ran two desktop benchmarks from PCMark05: the general HDD benchmark and the file decryption benchmark [112]. To simulate a desktop application workload, we also ran the FutureMark Peacekeeper HTML5/Javascript browser benchmark with the Google Chrome web browser.[1] The mean and 95% confidence interval of the mean for the percent overhead of running 30 samples of each benchmark are plotted in Fig. 9.3.

From Fig. 9.3, we see that the overhead from the keylogger detector in those

[1] http://peacekeeper.futuremark.com/

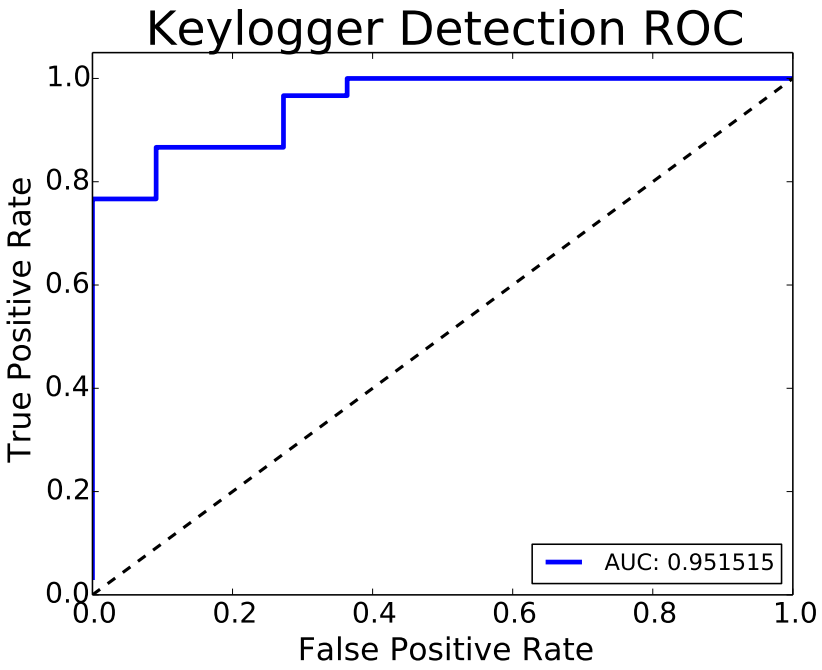


Figure 9.2: Receiver operating characteristics for the keylogger detector on Windows 7. The area under the curve was 0.95 (1.0 would represent a perfect system).

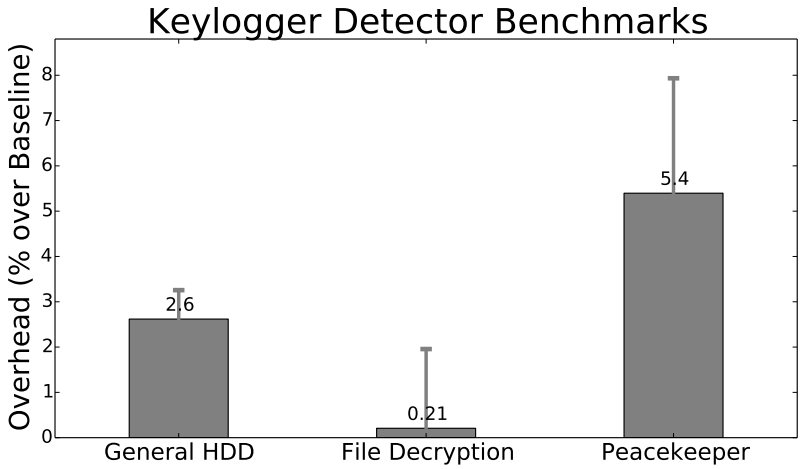


Figure 9.3: Benchmarks of the keylogger detector running on a Windows 7 VM.

experiments is between 0% and %, with a large dispersion in the Peacekeeper browser benchmark. We note that while representing a low overhead, these performance numbers reflect an unrealistic worst-case scenario. The primary cause of overhead during these benchmarks was re-enabling and logging all `CR3` VM Exits. As a performance improvement for production systems we could disable `CR3` VM Exits in between keystrokes after a certain threshold (e.g., when the contribution to R_k would be negligible). However, doing so would make benchmarking difficult as the performance impact would be more dependent on the typing rate of the user, and at the time of writing we did not have the ability to conduct performance tests involving users. We also note that for the VDI use case, performance is not critical as long as the system remains interactive since VDI deployments are not usually targeted for graphics-intensive workloads.

9.5 Discussion

From Fig. 9.2, we see that the Keylogger detector has an optimal False Positive Rate of 0 for a threshold of mean $\bar{R} = 1.69$ yielding a True Positive Rate of 0.77. Since keyloggers are usually long-lived processes, we can take this conservative value and expect to detect the keylogger eventually. We note that we have only thoroughly evaluated the keylogger detection for Windows 7. For other OSs, both the threshold $\bar{R}$ and $t_{1/2}$ will likely change. However, using QEMU's sendkey features, the parameter learning and tuning can occur outside of the VM, and therefore our abstraction of leaving the guest OS untouched remains unbroken.

One limitation of this keylogger monitor is that it only detects process-based keyloggers. There are indeed other classes of keylogger software (e.g., kernel-based or DLL based), but process based are the most common and easiest to deploy.

Chapter 10

Conclusions

In this dissertation, we presented a VM monitoring system that is supported by dynamic hook-based monitoring. We presented a novel way for addressing the semantic gap by bypassing that gap through the use of OS constructs. Using the dynamic monitoring capabilities of hprobes and the ability to monitor while leaving a cloud user's VM undisturbed, we find this system suitable for as-a-service reliability and security monitoring. In order to demonstrate the range of functionality that could be provided using our VM monitoring research, we presented multiple sample detectors that range from hang and infinite loop detection to keylogger detection for Virtual Desktop environments.

The hprobe framework uses the `int3` instruction, a simple and robust hook-based VM monitoring technique that does incur VM Exit overhead. For environments that have more stringent performance requirements, one could use an in-VM hook-based monitoring platform, and further research is needed to design a system to do so without requiring in-VM modifications [7, 109]. We stress that our RSaaS concept is independent of the specific hooking mechanism as long as the hooks can be set to trap on arbitrary addresses inside the VM. We chose VM Exits for the security properties and small attack surface that they offer.

In the full RSaaS system, the parameter inference and runtime monitoring steps are decoupled. As such, there is a certain level of trust in the guest OS. If the integrity of the VM is not protected, an attacker could avoid monitors by modifying the kernel code. Many of our detectors are designed to detect attacks before those attacks modify the kernel level, so the effort required for modifying kernel code has been increased when running our monitors. Additionally, in many cases the monitoring hooks are triggered by expected guest OS events and the absence of events from those actions could be a sign that the system is under attack.

The parameter inference step can be run offline, on a copy of the VM image, or online with a QEMU copy-on-write fork of the running VM image. If needed, the DAF's profiling can be based on the user's workload if it is configured to start at boot. Otherwise, one could live-migrate the VM after starting the workload to the emulator-based analysis environment for a brief profiling period [113]. Note that using snapshotting or live migration, one can infer parameters that change across boot by performing dynamic analysis on every VM startup (e.g., in the presence of ASLR or drivers being loaded in non-deterministic order).

While our targeted application domain was an Infrastructure as a Service cloud, our technique has broader applicability. In particular, large-scale virtualized environments are common in enterprise IT. In 2013, of 1750 IT leaders surveyed, only 5% of the respondents were not using virtualization (1% were explicitly not using virtualization, 4% did not know) [114]. Our runtime adaptable approach to monitoring is amenable to enterprise IT systems as those systems often have stringent uptime requirements and cannot restart systems to integrate new monitoring functionality.

Our prototype was built on open-source software, but our technique is compatible with closed-source commercial solutions. Tools exist to convert commercial VM image formats to the formats supported and QEMU-based DAFs. VMware offers its own hook-based VM monitoring solution [52]. Therefore, all the technology needed to port the presented framework to a commercial system like VMware already exists.

10.1 Future Work

An important continuation of this work would be implementing interfaces for the provider and customer and defining service-level agreements (SLAs) to address what guarantees a customer can expect from the provider. The provider could also use existing research on optimal monitor placement to offer insight on which systems to monitor in a multi-VM architecture [115–117].

When considering future architectures, it is important to note that the properties of the detectors presented in this dissertation are based on general systems and architectural principles. The hypervisor was used for monitoring

since it is an integral component of the cloud and has strong isolation properties, but the monitoring techniques presented in this dissertation could be used in a lightweight environment that only uses HAV for monitoring [45,46]. Furthermore, these monitoring techniques are fully applicable to containers [118] and future work should consider how to provide monitoring based on the unique properties of container environments.

Appendix A

Performance Comparison of Virtualization Technologies

A.1 Motivation

This appendix largely comprises work from [119]. In that work, we explored the performance cost of virtualization for sequence alignment software in genomics. Sequence alignment offers an interesting use case for testing performance as scientific applications are often considered too high-performance to pay the cost of virtualization. In the full paper, we studied both the Burrows-Wheeler Aligner [120] (BWA) and Novoalign [121]. In this section, we present the results from BWA as it had a more significant response and dependence on the virtualization technology.

A.2 Virtualization Technologies Tested

In addition to testing the performance of KVM, the virtualization technology of choice for this thesis, we tested two other technologies that are often used in cloud services: Xen and Linux Containers.

Xen is considered a full Type I hypervisor; even the "host" OS that has direct access to the hardware and manages the other VMs runs inside a special VM called "Domain 0." Xen helped popularize the concept of paravirtualization, where a modified version of the operating system is run to avoid the unnecessary redundancy and overhead that come with providing a full system (e.g., having to virtualize x86 instructions that are difficult to virtualize) to the guest OS [14]. Xen is an important use case because the most widely used public cloud IaaS platform, Amazon EC2, is built on top of a modified version of Xen. While the results presented here may not be representative of performance on EC2, Xen is still a popular hypervisor.

Linux Containers (LXC) represent a different virtualization paradigm that offers less isolation than a VMM. All containers share the same operating system kernel [118]. The advantage of sharing an OS kernel means that there is no device emulation and no duplicated effort of running an OS on top of another OS. This method is not virtualization in the usual sense, but containers can be used for many of the same reasons one would use virtual machines (e.g., isolating applications and offering on-demand access to a set of computing resources). The isolation provided by LXC is stronger than usual OS process isolation, thanks to the use of the cgroups [122] and kernel namespace features in Linux [123]. Kernel namespaces allow one to have separate domains for certain kernel objects (i.e. processes, users, network interfaces, and mounts) and cgroups allow one to allocate and partition resources among different processes. The most popular and successful example of LXC is in the open-source Docker project [124].

A.3 Experimental Setup

The experiments were performed on two classes of machines, a Dell PowerEdge R720 server with dual-socket 8 core Intel Xeon E5-2660 "Sandy Bridge" 2.20GHz CPUs (3.0GHz Turbo boost) with 20MiB of cache and a homebuilt workstation computer with dual-socket 6 core Intel Xeon E5645 "Westmere" 2.40GHz CPUs (2.67 GHz Turbo boost) with 12MiB cache. Since multiple machines were available, experiments were repeated on two identical Dell R720 machines to rule out machine-specific bias and there was no discernible difference between the two. The Dell servers had 128GiB of DDR3-1333MHz memory and 8 1TiB ST91000640SS 6.0Gb/s SAS drives arranged in a RAID 1+0 array. The RAID controller used (PERC H710P Mini) has a 1024MiB battery backed cache, with the disk array set to the default 'write back' mode. This server configuration represents a typical enterprise class large-memory server one could expect to find in a data center or research cluster. The workstation has 32GiB of DDR3-1333Mhz memory with a single 1TiB ST1000DM003 6.0Gb/s SATA hard drive. Throughout this appendix, the Sandy Bridge Dell servers will be referred to as "Machine 1" and the homebuilt Westmere workstation as "Machine 2." The hardware specifications of each machine are summarized in Table A.1.

Table A.1: Experimental Setup for Hypervisor Performance

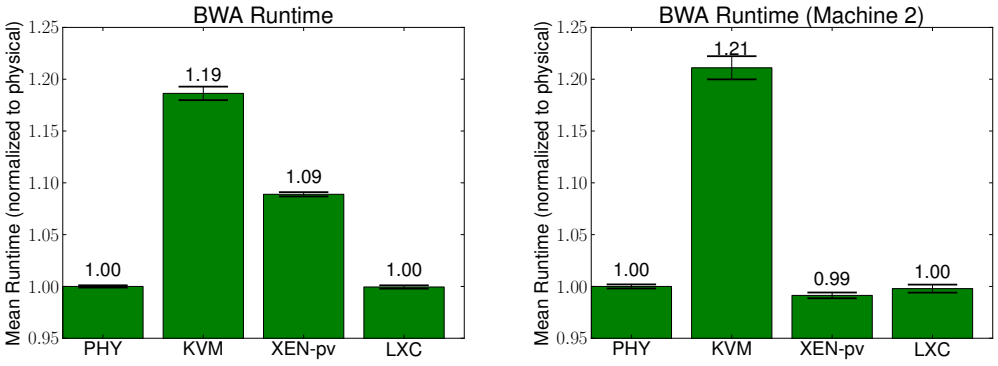
Machine	System Name	CPU	Memory	Storage
Machine 1	Dell PowerEdge R720	Dual-socket Intel Xeon E5-2660	128GiB DDR3-1333MHz	8x1TiB 6.0Gb/s SAS RAID 1+0
Machine 2	"Homebuilt" workstation	Dual-socket Intel Xeon E5645	32GiB DDR3-1333MHz	1TiB 6.0Gb/s SATA

The libvirt API was used to configure and manage VMs for all platforms. The VMs started as OpenStack instances based on the 12.04 LTS Ubuntu cloud image.[1] Both of the CPUs in the different machines support Intel VT-x HAV technology with VT-d, and have support for EPT. Each VM was allocated 8GiB of memory and 4 vCPUs.

Scientific applications use a large amount of input data from the disk, and it is important to account for how this input data is cached. If one is using virtual machine images that are layered on top of an existing filesystem (as could be expected in a cloud environment utilizing the KVM hypervisor), then both the VM operating system and host operating system could be caching data, depending on how the VMM is configured. Therefore, it is important to clear the buffer cache between all data samples to ensure that each experiment starts in a similar state (one could also reboot the machines to ensure a consistent state, but our experimentation has shown this to be excessive). In these experiments, virtual machines were restarted and the buffer cache was cleared before each sample.

Most modern multi-processors use a Non-Uniform Memory Access (NUMA) architecture. In a NUMA architecture, different processors/cores communicate with different regions of memory at different speeds, with fastest access to "local" memory and slower access to "non-local" memory (typically on a different socket). For optimal performance, a process should run on a CPU that has local access to the memory that process is accessing. Most modern OS schedulers are aware of NUMA, but one can still direct the OS to only run a process on a certain set of CPUs by "pinning" the process to those

[1] http://cloud-images.ubuntu.com/precise/



(a) (b)

Figure A.1: Results from 30 samples (10 on Machine 2) of running BWA on simulated paired-end Chromosome 1 reads on one virtual machine with the hypervisors in default configuration for Ubuntu 12.04. The x-axis indicates the hypervisor and the error bars indicate the 95% confidence interval. **(a)** Machine 1 (Sandy Bridge R720), where the physical server took a mean of 1955.49 seconds to complete the alignment. **(b)** Machine 2 (Westemere workstation), where PHY took a mean of 2460.09 seconds to complete. (Note that even though XEN-pv has a normalized runtime of 0.99, the confidence intervals of PHY and XEN-pv overlap.)

CPUs (e.g. by using the `taskset` command in Linux) [125].

A.4 Initial Measurements

The results of running single-threaded BWA 30 times on all four execution environments (physical server, Kernel Virtual Machine, para-virtualized Xen, and Linux Containers) are shown in Fig. A.1a. In this test, the input data consisted of paired-end 75bp reads sampled uniformly from human chromosome 1 (sourced from the UCSC hg19 reference[2]) with 2x coverage using the ART Illumina read simulator [126]. Here, we see that LXC's performance closely matches that of the physical server, but Xen and KVM have measurable overhead, with KVM's being significant.

[2]http://hgdownload.soe.ucsc.edu/goldenPath/hg19/chromosomes/chr1.fa.gz

A.5 Tuning KVM for Performance

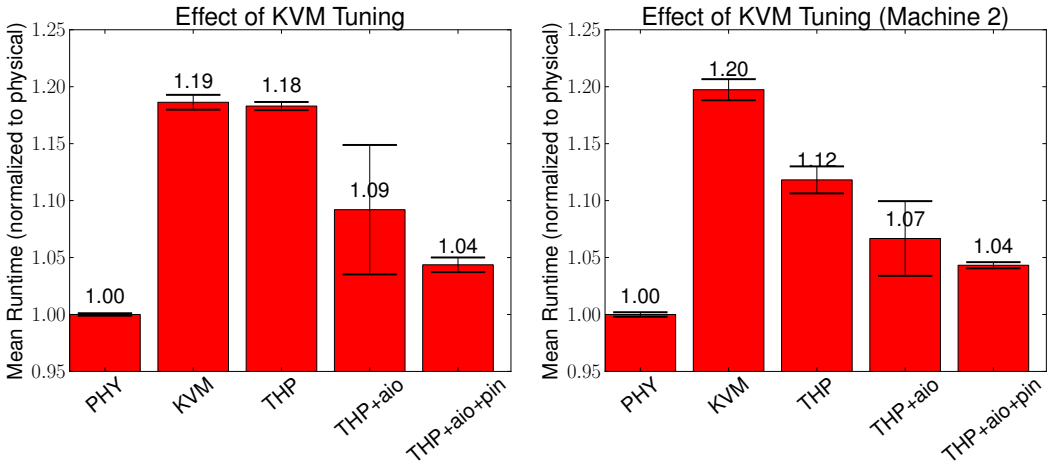
From Fig. A.1a, we saw that KVM had the high overhead of roughly 20% on both machines. To investigate the cause of this overhead, we start with recommended tuning practices for KVM [127]. The results of KVM tuning are summarized in Fig. A.2. As sequence aligners are memory-intensive applications, improving memory performance should have a positive effect on application performance. Furthermore, since these applications also work with large files, we also investigate parameters for tuning disk performance.

One technique to improve memory performance is to reduce the number of TLB misses by using "huge" pages, or memory pages larger than the system's default. Traditionally, huge pages would need to be reserved by the OS at boot time, but *transparent huge pages* (THP) can be dynamically assigned by the OS heap and stack space for an application. Data points designated with "THP" (see Fig. A.2) had THP enabled and we see that THP offers modest performance gains, depending on the machine type.

Though the paravirtualized `virtio` drivers were used for KVM (vs. full device emulation in QEMU), there was some tuning left. The `libvirt` API defaults to using QEMU worker thread pools to emulate an asynchronous I/O (AIO) system. On recent Linux kernels support for native AIO is offered.[3] Experiments using native Linux AIO are designated by "aio" in Fig. A.2. We see that using native AIO is helpful in reducing the mean time on both machines. In addition to AIO, we also tested different file caching strategies, but we found that the cache='none' chosen by OpenStack when creating the VM image to yield the best performance.

As mentioned earlier, modern CPUs (sockets) can have asymmetric memory access times to different memory regions. A specific region of memory (e.g., memory associated with CPU socket 0) is called a NUMA node. In our experiments, we observed that in the default configuration roughly half of a VM's pages would reside on one NUMA node, and the other half of a VM's pages would reside on the other node (both types of machines we used had two CPU sockets and therefore two NUMA nodes). In order to prevent processors from accessing memory non-local to their NUMA node, one can restrict virtual CPUs to a set of physical CPUs. In Fig. A.2, the "pin" data

[3] Seehttp://www.linux-kvm.org/page/Virtio/Block/Latencyandhttps://www.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaatbpkvmasynchio.htm



(a) (b)

Figure A.2: Results from 10 samples of running BWA on the KVM hypervisor with various tuning options. PHY indicates the physical server and KVM indicates KVM with a "default" configuration. THP signifies the hypervisor had Transparent Huge Pages enabled, aio signifies that the VM was using Linux native asynchronous I/O, and pin indicates the VM was pinned to CPUs on the same NUMA node. **(a)** Machine 1 (Sandy Bridge R720), where the physical server (PHY) took a mean of 1955 seconds to complete the alignment. **(b)** Machine 2 (Westemere workstation), where PHY took a mean of 2460 seconds to complete.

points refer to runs where vCPUs were manually pinned to a specific NUMA node.

We see that with all of these tuning parameters the performance overhead of KVM was brought down from 20% to 4%. There still may be other ways to increase the performance, but it is important to realize that the overhead of virtualization can be significantly dependent on not only the type of virtualization, but also the physical hardware and configuration of the VMM.

References

[1] P. Hernandez, "Skype, aws outages rekindle cloud reliability concerns," Online, http://www.eweek.com/cloud/skype-aws-outages-rekindle-cloud-reliability-concerns.html, 2015.

[2] T. Holdings, "Trustwave global security report," *Retrieved July*, vol. 26, p. 2016, 2015.

[3] T. Garfinkel, M. Rosenblum et al., "A virtual machine introspection based architecture for intrusion detection," in *NDSS*, vol. 3, 2003, pp. 191–206.

[4] B. D. Payne, M. De Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Proc. 23rd Ann. Computer Security Applications Conf. (ACSAC) 2007*. IEEE, 2007, pp. 385–397.

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

[6] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2008, pp. 91–100.

[7] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *In Proc of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 477–487.

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

[9] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[10] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *In Proc. of the Fourth ACM Symposium on Operating System Principles*, pp. 121–, 1973.

[11] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The Turtles Project: Design and implementation of nested virtualization." in *OSDI*, vol. 10, 2010, pp. 423–436.

[12] R. P. Goldberg, "Architectural principles for virtual computer systems," Ph.D. dissertation, Harvard University, 1973.

[13] IBM Corporation, "z/VM - A brief review of its 40 year history," 2012. [Online]. Available: http://www.vm.ibm.com/vm40hist.pdf

[14] 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," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[15] E. Bugnion, "Dynamic binary translator with a system and method for updating and maintaining coherency of a translation cache," Mar. 9 2004, US Patent 6,704,925.

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

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

[18] N. Bhatia, "Performance evaluation of Intel EPT hardware assist," *VMware, Inc*, 2009.

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

[20] Linux Kernel Developers, "The definitive KVM (Kernel-based Virtual Machine) API documentation," 2016. [Online]. Available: https://kernel.org/doc/Documentation/virtual/kvm/api.txt

[21] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehörster, and A. Brinkmann, "Non-intrusive virtualization management using libvirt," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 574–579.

[22] P. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[23] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. IEEE, 2010, pp. 222–226.

[24] "CVE-2015-3456." Available from MITRE, CVE-ID CVE-2015-3456., Apr. 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456

[25] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman, "Delusional boot: Securing hypervisors without massive re-engineering," in *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, 2012, pp. 141–154.

[26] 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*. ACM, 2010, pp. 38–49.

[27] A. M. Azab, P. Ning, and X. Zhang, "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 375–388.

[28] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 616–630.

[29] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 437–450.

[30] Z. Liu, J. Lee, J. Zeng, Y. Wen, Z. Lin, and W. Shi, "CPU Transparent Protection of OS kernel and hypervisor integrity with programmable DRAM," *Proceedings of the 40th Annual International Symposium on Computer Architecture*, vol. 41, no. 3, 2013.

[31] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the 2013 International Workshop on Security in Cloud Computing*. ACM, 2013, pp. 3–10.

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

[33] K. Asrigo, L. Litty, and D. Lie, "Using VMM-based sensors to monitor honeypots," in *Proceedings of the 2nd International Conference on Virtual Execution Environments*. ACM, 2006, pp. 13–23.

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

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

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

[37] 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.

[38] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," Georgia Institute of Technology, Tech. Rep., 2011.

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

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

[41] G. Wang, Z. J. Estrada, C. Pham, Z. Kalbarczyk, and R. K. Iyer, "Hypervisor introspection: A technique for evading passive virtual machine monitoring," in *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

[42] P. Li, D. Gao, and M. Reiter, "Mitigating access-driven timing channels in clouds using stopwatch," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.

[43] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: Tracking processes in a virtual machine environment," in *USENIX Annual Technical Conference, General Track*, 2006, pp. 1–14.

[44] 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*. ACM, 2008, pp. 51–62.

[45] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

[46] K.-l. Tseng, "Intel kernel guard technology," Online, https://01.org/intel-kgt, 2015.

[47] 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.

[48] T. R. Martin, "Method for collecting ECC event-related information during SMM operations," Oct. 26 1999, US Patent 5,974,573.

[49] A. Borisov, "Coreboot at your service!" *Linux Journal*, vol. 2009, no. 186, p. 1, 2009. [Online]. Available: http://www.linuxjournal.com/magazine/coreboot-your-service

[50] B. Delgado and K. L. Karavanic, "Performance implications of system management mode," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 163–173.

[51] J. Masters, "[RFC] simple SMI detector," Online, https://lwn.net/Articles/316622/, 2009.

[52] M. Carbone, A. Kataria, R. Rugina, and V. Thampi, "Vprobes: Deep observability into the ESXi hypervisor," *VMware Technical Journal*, vol. 14, no. 5, pp. 35–42, 2014.

[53] Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer, "Dynamic VM dependability monitoring using hypervisor probes," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*. IEEE, 2015, pp. 61–72.

[54] T. Ball, "The concept of dynamic analysis," in *Software EngineeringESEC/FSE99*. Springer, 1999, pp. 216–234.

[55] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[56] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2.

[57] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[58] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Information Systems Security*. Springer, 2008, pp. 1–25.

[59] 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 & Communications Security*. ACM, 2013, pp. 839–850.

[60] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 248–258.

[61] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, "MAVMM: Lightweight and purpose built VMM for malware analysis," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 441–450.

[62] Y. Dong, Z. Yu, and G. Rose, "SR-IOV networking in Xen: Architecture, design and implementation," in *Workshop on I/O Virtualization*, 2008.

[63] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, "Down to the bare metal: Using processor features for binary analysis," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 189–198.

[64] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 569–584.

[65] K. Kourai and S. Chiba, "Hyperspector: Virtual distributed monitoring environments for secure intrusion detection," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. ACM, 2005, pp. 197–207.

[66] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A survey of intrusion detection techniques in cloud," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 42–57, 2013.

[67] P. Cao, E. Badger, Z. Kalbarczyk, R. Iyer, and A. Slagell, "Preemptive intrusion detection: Theoretical framework and real-world measurements," in *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. ACM, 2015, p. 5.

[68] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

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

[70] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Oct. 2014, pp. 249–265.

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

[72] 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.

[73] F. C. Eigler and R. Hat, "Problem solving with systemtap," in *Proc. of the Ottawa Linux Symposium*. Citeseer, 2006, pp. 261–268.

[74] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch, "Virtualize everything but time," in *OSDI*, vol. 10, 2010, pp. 1–6.

[75] P. Luszczek, E. Meek, S. Moore, D. Terpstra, V. M. Weaver, and J. Dongarra, "Evaluation of the HPC challenge benchmarks in virtualized environments," in *Euro-Par 2011: Parallel Processing Workshops*. Springer, 2012, pp. 436–445.

[76] 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.

[77] N. A. Quynh and K. Suzaki, "Xenprobes, a lightweight user-space probing framework for Xen virtual machine," in *USENIX Annual Technical Conference Proceedings*, 2007.

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

[79] 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.

[80] 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.

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

[82] 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.

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

[84] J. Corbet, "vmsplice(): the making of a local root exploit," Online, http://lwn.net/Articles/268783/, 2008.

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

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

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

[88] W. Lee, S. J. Stolfo, and P. K. Chan, "Learning patterns from Unix process execution traces for intrusion detection," in *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997, pp. 50–56.

[89] 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.

[90] S. J. Vaughan-Nichols, "No reboot patching comes to Linux 4.0," Online, http://www.zdnet.com/article/no-reboot-patching-comes-to-linux-4-0/, 2015.

[91] J. Corbet, "A rough patch for live patching," Online, http://lwn.net/Articles/634649/, 2015.

[92] J. Corbet, "Compile-time stack validation," Online, https://lwn.net/Articles/658333/, 2015.

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

[94] 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," *arXiv:0901.0866*, 2002.

[95] M. Hamdaqa and L. Tahvildari, "Cloud computing uncovered: a research landscape," *Advances in Computers*, vol. 86, pp. 41–85, 2012.

[96] S. J. Vaughan-Nichols, "Ubuntu Linux continues to rule the cloud," Online, http://www.zdnet.com/article/ubuntu-linux-continues-to-rule-the-cloud/, 2015.

[97] I. D. Craig, *Formal Models of Operating System Kernels*. Springer Science & Business Media, 2007.

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

[99] J. Clark, "5 numbers that illustrate the mind-bending size of amazon's cloud," Bloomberg Business, http://www.bloomberg.com/news/2014-11-14/5-numbers-that-illustrate-the-mind-bending-size-of-amazon-s-cloud.html, 2015.

[100] S. Siddha, V. Pallipadi, and A. Ven, "Getting maximum mileage out of tickless," in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 201–207.

[101] J. Corbet, "(Nearly) full tickless operation in 3.10," Online, http://lwn.net/Articles/549580/, 2013.

[102] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *USENIX Security*, vol. 3, 2003, pp. 105–120.

[103] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on.* IEEE, 2003, pp. 260–269.

[104] S. Keil and C. Kolbitsch, "Kernel-mode exploits primer," International Secure Systems Lab (isecLAB), Tech. Rep., 2007.

[105] D. Rosenberg, "SMEP: What is it, and how to beat it on Linux," Online, http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/, 2011.

[106] A. Shishkin and I. Smit, "Bypassing Intel SMEP on windows 8 x64 using return-oriented programming," Online, http://blog.ptsecurity.com/2012/09/bypassing-intel-smep-on-windows-8-x64.html, 2012.

[107] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on.* IEEE, 2014, pp. 491–502.

[108] D. W. Hill and J. T. Lynn, "Adaptive system and method for responding to computer network security attacks," July 11 2000, US Patent 6,088,804.

[109] S. Panneerselvam, M. Swift, and N. S. Kim, "Bolt: Faster reconfiguration in operating systems," in *2015 USENIX Annual Technical Conference (USENIX ATC 15).* Santa Clara, CA: USENIX Association, July 2015. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/panneerselvam pp. 511–516.

[110] S. Ortolani, C. Giuffrida, and B. Crispo, "Bait your hook: A novel detection technique for keyloggers." in *RAID.* Springer, 2010, pp. 198–217.

[111] J. Nielsen, "Response times: The 3 important limits," *Usability Engineering*, 1993.

[112] S. Niemela, "PCMark05 PC performance analysis," white paper from FutureMark Corp, 2005.

[113] J. Wei, L. K. Yan, and M. A. Hakim, "Mose: Live migration based on-the-fly software emulation," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2818000.2818022 pp. 221–230.

[114] C. McLellan, "Virtualizing the enterprise: An overview," Online, http://www.zdnet.com/article/virtualizing-the-enterprise-an-overview/, 2013.

[115] C. Chaudet, E. Fleury, I. G. Lassous, H. Rivano, and M.-E. Voge, "Optimal positioning of active and passive monitoring devices," in *Proceedings of the 2005 ACM conference on Emerging Network Experiment and Technology*. ACM, 2005, pp. 71–82.

[116] A. W. Jackson, W. Milliken, C. Santiváñez, M. Condell, W. T. Strayer et al., "A topological analysis of monitor placement," in *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*. IEEE, 2007, pp. 169–178.

[117] N. Talele, J. Teutsch, R. Erbacher, and T. Jaeger, "Monitor placement for large-scale systems," in *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*. ACM, 2014, pp. 29–40.

[118] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 275–287.

[119] Z. J. Estrada, F. Deng, Z. Stephens, C. Pham, Z. Kalbarczyk, and R. Iyer, "Performance comparison and tuning of virtual machines for sequence alignment software," *Scalable Computing: Practice and Experience*, vol. 16, no. 1, pp. 71–84, 2015.

[120] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[121] "Novocraft technologies," Online, http://novocraft.com, 2014.

[122] P. Menage, "CGROUPS," Online, https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt, 2006.

[123] M. Kerrisk, "Namespaces in operation, part 1: namespaces overview," Online, http://lwn.net/Articles/531114/, 2013.

[124] S. Hykers, "What is Docker?" Online, https://www.docker.com/whatisdocker/, 2016.

[125] R. Love, "CPU affinity," *Linux Journal*, no. 111, July 2003. [Online]. Available: http://www.linuxjournal.com/article/6799

[126] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "ART: A next-generation sequencing read simulator," *Bioinformatics*, vol. 28, no. 4, pp. 593–594, 2012.

[127] M. Wagner, "KVM performance improvements and optimizations," Online, http://www.linux-kvm.org/wiki/images/5/59/Kvm-forum-2011-performance-improvements-optimizations-D.pdf, 2011.