Hypervisor-Based System Call Introspection with Extended Berkeley Packet Filter

by

Huzaifa Patel

A thesis proposal submitted to the School of Computer Science in partial fulfillment of the requirements for the degree of

Bachelor of Computer Science

Under the supervision of Dr. Anil Somayaji

Carleton University

Ottawa, Ontario

September, 2022

Abstract

Soon

Acknowledgments

I want to express my heartfelt gratitude to my supervisor, Dr. Anil Somayaji for providing me with the opportunity to work on a thesis during the final year of my undergraduate degree. Unlike previous variations of the Computer Science undergraduate degree requirements, completing a thesis is no longer a prerequisite. Therefore, I prostualte it is a great privlidge and honor to be given the opportunity to enroll into a thesis-based course during ones undergraduate studies.

I did not have prior experience in formal research when I first approached Dr. Somayaji. Despite this shortcoming, it did not stop him from investing his time and resources towards my academic growth. Without his feedback and ideas on my framework implementation and writing of this thesis, as well as his expertise in eBPF, Hypervisors, and Unix based Operating Systems, this thesis would not have been possible.

I would like to commend PhD student Manuel Andreas from The Technical University of Munich, Germany for introducing me to the concept of a Hypervisor. Without him, I would not have approached Dr. Somayaji with the intention of wanting to conduct research on them. His minor action of introducing me to hypervisors had the significant effect of inspiring me to write a thesis on the subject. I also want to thank him for his willingness to endlessly and tirelessly teach, discuss and help me understand the intricacies of hypervisors, the Linux kernel, and the C programming language.

I would also like to thank Carleton University's faculty of Computer Science for their efforts in imparting knowledge that has enthraled and inspired me to learn all that I can about Computer Science.

I would like to extend my appreciation to the various internet communities which have provided the world with invaluable compiled resources on hypervisors, Unix based operating systems, eBPF, the Linux kernel, the C programming language, and Latex, which has helped me tremendously in writing this thesis.

Finally, I would like to thank my immediate family for their encouragement and support towards my research interests and educational pursuits.

Contents

List of Figures

List of Tables

Listings

Nomenclature

VM	Virtual Machine
KVM	Kernel-based Virtual Machine
OS	Operating System
VMI	Virtual Machine Introspection
CPU	Central Processing Unit
AMD	Advanced Micro Devices
AMD-V	Advanced Micro Devices Virtualization
VT-x	Intel Virtualization Extension
VMX	Virtual Machine Extensions, analogous to VT-x
MSR	Model Specific Register
VMM	Virtual Machine Monitor, analogous to a hypervisor
EFER	Extended Feature Enable Register
eBPF	Extended Berkeley Packet Filter
VMI	Virtual Machine Introspection
API	Application Programming Interface
IDS	Intrusion Detection System
JIT	Just-in-time
MMU	Memory Management Unit
QEMU	Quick Emulator
GPF	General Protection Fault
IEEE	Institute of Electrical and Electronics Engineers
GDB	GNU Debugger
NMI	Non-maskable Interrupt
TCG	Tiny Code Generator
BCC	BPF Compiler Collection
SMAP	Supervisor Mode Access Prevention
KPTI	Kernel page-table isolation
HCI	Human Computer Interaction
IPS	Intrusion Prevention System
TLB	Translation Look-aside Buffer

Introduction

Cloud computing is a modern method for delivering computing power, storage services, databases, networking, analytics, and software applications over the internet (the cloud). Organizations of every type, size, and industry are using the cloud for a wide variety of use cases, such as data backup, disaster recovery, virtual desktops, software deployment, testing, big data analytics, and web applications [3]. For example, healthcare companies use the cloud to store patient records in databases [3]. Financial service companies use the cloud for real-time fraud detection and prevention in order to save client assets [3]. And finally, video game companies use the cloud to deliver online video game services to millions of players around the world.

The existance of cloud computing can be attributed to virtualization, which is a technology that makes it possible for multiple different operating systems (OSs) to run concurrently, and in an isolated environment on the same hardware. How does virtualization achieve this? It makes use of a machines hardware to support the software that creates and manages virtual machines (VMs). A VM is a virtual environment that provides the functionality of a physical computer by using its own virtual central processing unit (VCPU), memory, network interface, and storage. The software that creates and manages VMs is formally called a hypervisor or virtual machine monitor (VMM). The virtualization marketplace is comprised of four notable hypervisors, which are: (1) VMWare, (2) Xen, (3) Kernel-based Virtual Machine (KVM), and (4) Hyper-V. The operating system running a hypervisor is called the host OS, while the VM that uses the hypervisor is called the guest OS. To maintain consistency within this proposal, we will use the word "hypervisor" when referring to virtualization software.

While virtualization technology can be sourced back to the 1970s, it wasn't widely

adopted until the early 2000s due to hardware limitations [24]. The fundamental reason for introducing a hypervisor layer on a modern machine is that without one, only a single operating system would be able to run at a given time. This constraint often led to wasted resources, as a single OS infrequently utilized the full capacity of modern hardware. More specifically, the computing capacity of a modern CPU is so large, that under most workloads, it is difficult for a single OS to efficiently use all of its resources at a given time. Hypervisors address this constraint by allowing all of a system's resources to be used by distributing them over several VMs. This allows users to switch between one machine, many operating systems, and multiple applications at their discretion.

1.1 The Problem

Modern computer hardware has made our systems faster and larger, which has allowed for hundreds of processes to run concurrently on any given VM. Each of these processess contain a remarkable amount of complexity and functionality, and each of the executables that the process contains, requires tens of megabytes of memory and hundreds of megabytes of disk space [25]. A result of our computer systems becoming increasingly complex is that they have become more unpredictable and unreliable. For example, new vulnerabilities are discovered almost every day on major OSs, and in both native and 3rd party applications. When these vulnerabilities are addressed with software updates, it is not uncommon for new ones to be discovered soon after [25]. Furthermore, the continual requirement for VMs to be connected to local networks and/or the Internet further reduces their reliability and security due to the ability for large amounts of unfiltered nondeterministic data to enter a VM from these networks. [25]. The need for connectivity has made local networks and the Internet a common vector for attackers who want to access or manipulate a virtualized environment [29]. As such, a VM that is isolated may behave consistently over time, whereas one that is connected to the internet will not [25]. For the reasons mentioned above, the role of a VM is highly security critical, and thus, one of its priorities should be to maintain confidentiality, integrity, authorization, availability, and accountability throughout its life cycle [27]. A successful attack on a VM can result in the violation of one or more of

the aforementioned goals of computer security. For example, an attack can result in the loss of availability of services due to a denial-of-service attack. Secondly, an attacker can make private information accessible to unauthorised parties. Additionally, data, software or hardware can be altered by unauthorized parties. And worst of all, malicious actions committed by an attacker can go unnoticed due to a system not adopting measures to properly track user actions (repudiation) [27]. From what was mentioned in our introduction, it is evident that virtualization provides a whole new standard concerning cloud computing, and its ability to provide cheaper computing services to the world. However, with these added benefits, threat management is still a topic of concern. For these reasons, effective methodologies for monitoring and responding to VM anomalies is required. What follows is a brief introduction on how we decide on a methodology, and how our decision will help us address the problem of maintaining the fundamental goals of computer security on particular VMs.

1.2 Addressing the Problem

As computer systems become more complex, it becomes more difficult to determine exactly what they are doing at any given time. Modern computers run dozens, if not hundreds of processes at once, the vast majority of which run silently in the background [9]. This begs the question: How do we determine the best methodology for monitoring VMs for anomalies? We could place all the responsibility on the user for the safety of their virtualized environment. However, this is not practical because users have a limited idea of what is happening on their systems, especially beyond what they can see on their screens [9]. Fundamentally, there is too much occuring on a computer system, which makes a human unable to deduce whether their system is misbehaving at any given time [9]. If users are not good candidates for adequately monitoring VMs for malicious anomalies, then the better option is to let a computer system implictly watch over itself with the help of a virtual machine introspection (VMI) system. In computing, VMI is a technique for monitoring and sometimes responding to the runtime state of a VM based on certain predefined conditions [20]. However, current Linux systems do not have a native VMI system. Thus, one must build or install such a system on their machine.

In this thesis, we present Frail, a hypervisor-based virtual machine introspection system that is exclusive to (1) Linux kernel versions 5.18.8+, (2) the KVM hypervisor, (3) and Intel Virtualization Extension (VT-x). Our VMI is intended to enhance some of the capabilities of a previously developed KVM VMI system called Nitro. More specifically, Frail is a VMI that is intended allow one to (1) trace KVM guest system calls and their corresponding processes, (2) monitor these system calls and processes for anomalies, and (3) respond to these anomalies from the hypervisor level. Our framework is implemented using a combination of existing and our own software. Firstly, it utilizes a custom Linux kernel, custom KVM module, and a Linux native program called Extended Berkeley Packet Filter (eBPF) to safely tarce both KVM guest system calls and the corresponding KVM guest process that requested the system call. Secondly, it uses Dr. Somayaji's implementation of sequences of system calls (pH) to detect malicious anomalies [25]. Lastly, we respond to the anomalous system calls (with our own implementation) by either terminating the VM that the anomalous system call came from, or terminating the guest process that was deemed responsible for the observed anomaly. Our intention is to make it possible for our VMI system to trace, monitor, and respond in real-time without hindering the usability of the guest and host. To our knowledge, Frail is the second KVM hypervisor-based VMI system that is intended to support the tracing and monitoring of system call instructions provided by modern Intel x86 architectures (SYSCALL, SYSRET, SYSENTER). Likewise, to our knowledge, Nitro is the first KVM hypervisor-based VMI system that is intended to utilize sequences of system calls to monitor anomalies, and respond to them in the way we mentioned above.

1.3 The Problem (Continued): Research Questions

Although hypervisor-based VMI systems have many advantages over a user monitoring their virtualized environment, there still exists many challenges in the design, implementation, and deployment stages of its development. This proposal aims to address the following six challenges posed by hypervisor-based VMI systems in Chapter Three, one at a time:

Research Question 1: KVM is formally defined as a type 1 hypervisor by its creators RedHat. As a result, guest instructions that are defined by the CPU are sent directly to the CPU. System calls (SYSCALL, SYSRET, SYSENTER) are an example of instruction that Intel x86 CPUs define. Can we change the route of system calls so that they are trapped and emulated at the hypervisor level instead of being sent directly to the CPU for execution?

Research Question 2: Can we effectively retrieve KVM guest system calls and the the process that requested the system call from the guest by bridging the semantic gap of the KVM hypervisor?

Research Question 3: Can we make use of KVM guest system calls and sequences of system calls to successfully detect anomalies in real-time with a high success rate, and without hindering the usability of the guest and host?

Research Question 4: What extensions to the Linux tracepoints API would be required for eBPF to successfully trace KVM guest system calls and the guest process that requested the system call?

Research Question 5: Can we effectively terminate an anomalous guest process or the VM by bridging the semantic gap of the KVM hypervisor?

Research Question 6: Can we deploy our hypervisor-based VMI system without hindering the confidentially, integrity, authorization, availability, and accountability of both the host and guest?

1.4 Motivation

In this section, we comprehensively explain our reasoning/motivation for designing our VMI system in the manner that we did.

1.4.1 Why Design a New VMI?

The topic of securing VMs dates back to 2003, when Tal Garfinkel and Mendel Rosenblum proposed VMI as a hypervisor-based intrusion detection system (IDS) that integrated the benefits of both network-based and host-based IDS (see section 2.9) [13] [25]. Since then, widespread research and development of VMs has led to an abundance in VMI systems, some more practical than others, but all for the purpose of monitoring VMs. What follows is a discussion as to why we believe it is necessary to design and implement yet another VMI system, despite the fact that many already exist.

At the time of writing this thesis, to our knowledge, there is one relevant and related KVM VMI system named Nitro. More specifically, Nitro is a VMI system for system call tracing and monitoring, which was intended, implemented, and proven to support Windows, Linux, 32-bit, and 64-bit environments [22]. The problem with Nitro is that it is now over 11 years old, and its official codebase has not been updated in over 6 years. For this reason, it is no longer compatible with any Linux 32-bit and 64-bit environments, and is not compatiable with newer Windows desktop versions. In fact, at the time of writing this proposal, Nitro only supports Windows XP x64 and Windows 7 x64, which makes it ineffective for use on newer computer systems. Also, it is impractical to use it on supported OSs for two reasons. Firstly, Windows XP and Windows 7 are discontinued OSs, which means that security updates and technical support are no longer available. Secondly, Windows XP is over 21 years old and as of July 2021, it consists of only 0.59% of the marketshare of worldwide Windows desktop versions running [17]. Similarly, Windows 7 is 13 years old, and consists of only 16.06% of the same marketshare. [17].

There is a fundamental problem with the state of many existing VMI's like Nitro: when the codebase of either an OS or the kernel changes, VMI's are unable to solve the problem for which they were originally designed to solve: to trace and monitor VMs that are running Windows, Linux, 32-bit, and 64-bit environments [29]. The primary reason for this problem is that VMIs were designed in such a way that compromised compatibility and adaptability with subsequent versions of the OSs with which they were originally intended, implemented, and proven to be compatible with. To solve the problem of incapability, we seek to design a spiritual successsor to Nitro that is intended to provide

a VMI without sacrificing compatibility with subsequent versions of the Linux kernel.

1.4.2 Why Design a Hypervisor-Based VMI System?

A VMI system can either be placed in each VM that requires monitoring (guest-based VMI), or it can be placed on the hypervisor level outside of every VM (hypervisor-based VMI). In this subsection, we justify our motivations for designing and implementing a hypervisor-based VMI by analyzing the advantages and disadvantages of both types of VMI systems. We begin by discussing the four key advantages of hypervisor-based VMI's: isolation, inspection, interposition, and deployability [21].

In our context, isolation refers to the property that hypervisor-based VMI systems are tamper-resistant by the VMs that are being monitored. Tamper resistant in our context, is the property that VMs are unable to commit unauthorized access or alteration of any component of a VMI system (i.e. code, stored data, and more). If we assume that a hypervisor is free of vulnerabilities, then both the hypervisor and hypervisor-based VMI system is considered isolated from every guest VM. The implication above holds true because hypervisor-based VMIs run at a higher privlige level than guests. [13]. Therefore, if a hypervisor is free of vulnerabilities, then guest VMs will not have a way of attacking the hypervisor or the hypervisor-based VMI system. Guest-based VMI systems are unable to maintain the property of isolation because they are deployed in the guest VM. Thus, by default, they are vulnerable to being altered by the VMs that are being monitored by the VMI system. When the property of isolation holds for a hypervisor-based VMI, there exists two key advantages. Firstly, if a hypervisor is managing a set of VMs, it is possible for a subset of those VMs to be considered untrusted due to a successful attack from within their confined environment. If a hypervisor-based VMI holds the property of isolation, then both the hypervisor-based VMI system and hypervisor will be immune from attacks that originate in the guest, even if the VMI is actively monitoring a guest that is under attack [13]. Secondly, due to the isolation of hypervisor-based VMI's from the guest, the VMI only needs to trust the underlying hypervisor instead of the entire Linux kernel. In contrast, if a VMI was deployed in a guest, then the entire guest kernel would need to be trusted. Having to trust only the hypervisor is advantagous because the KVM hypervisor has less than one twelfth the

number of lines of code than the Linux kernel; this smaller attack surface leads to fewer vulnerabilities in hypervisor-based VMI systems [1].

Inspection refers to the property that allows a VMI system to examine the entire state of the guest while continuing to be isolated [21]. Hypervisor-based VMI's run one layer below all the guests, and on the same layer of the hypervisor. For this reason, the VMI is capable of having a complete view of all guest OS states (CPU registers, memory, and more) of every VM [13]. A VMI isolated from the VM also offers the advantage for a constant and consistent view of the system state, even if a VM is in a paused state. In contrast, a guest-based VMI system would stop executing when a VM enters a paused state.

Interposition is the ability to inject operations into a running VM based on certain conditions. Due to the close proximity of a hypervisor and a hypervisor-based VMI system, the VMI is capable of modifying any of the states of the guest and interfering with every activity of the guest. As a result, interposition makes it easy for our VMI system to respond to observed anomalies by terminating the guest process or VM that is responsible for the anomaly [13].

Deployability refers to how easily a VMI system can be moved from the development stage to full-scale deployment on a system. Deployability can be measured in terms of the number of discrete steps required to deploy a VMI system to the production environment. To deploy a hypervisor-based VMI system, no guest has to be modified to accomodate for the VMI's deployment. For example, we do not have to make a new user for any guest VM, nor do have to install the VMI system software or its dependencies in any of the guest VMs. Instead, we only need to install the VMI system and its dependencies on the host OS once. In contrast, a guest-based VMI system and its dependencies must be installed on each VM that requires monitoring, which necessitates many more steps than setting up a hypervisor-based VMI system.

Although guest-based VMI systems have been successful, they are more susceptible to two types of threats: (1) privilege escalation, and (2) tampering [21].

As previously mentioned, guest-based VMI systems are not isolated because they are executing on the same privilege level as the VMs that they are protecting [2]. As a result, malicious software (malware), such as kernel rootkits can be used to conduct privilege escalation. Privilege escalation is the act of exploiting a bug or a design flaw in an operating system or software application to gain elevated access to resources that are normally protected from an application or user. The result is that an application or user has more privileges than intended by the application developer or system administrator. After a successful privlige escalation, attackers can carry out unauthorised actions. For instance, the following scenarios are possible in the event of a successful privlige escalation:

- An attacker can tamper with the tracing software that collects system call information and/or process identification information.
- As our VMI depends on hooking specific kernel functions, attackers can modify the relevant symbols within the symbol table with a simple kernel module. In other words, they could hook their own function in place of our hooked function, which would allow them to bypass our VMI properties.
- Attackers can tamper with the pH, the program that utilizes sequences of system calls to monitor for anomalous system calls. In this scenario, attackers can prevent anomalous system calls from being declared, essentially allowing a repudiation attack because the VMI controls responsible for tracking anomalous system calls is now tampered with.
- The VMI system that responds to anomalous processes can be tampered with. Currently, our security policy consists of either terminating the VM or the anomalous process. Attackers can tamper our security policy so that the process that requested the anomalous system call is never terminated or the VM is never shut down.
- The log files that contain information about anomalous system calls, process information, and normal behavior can be tampered with by overwriting or appending

them with false data.

In all the cases above, as long as a successful attack results in the VMI to continue its execution (e.g., no crashes), the VMI system can be made to generate a false pretense to mislead security experts into thinking that a VM process is not malicious when it actually is.

Guest-based VMI's have two unique advantages: (1) rich abstractions, and (2) speed.

Because the user space fills the semantic gap by providing interfaces to extract OS level information, guest-based VMI's are able to trivially intercept system calls and process information. For example, we can use kernel variables and functions to trace system call and process information by using kprobes or the Linux Tracepoint API (see section 2.8). Even simpler, we can use third party Linux tools like strace to extract system calls to inspect their sequences. We can also trivially develop an IPS system that terminates an anomalous process by calling a lib C exit system call wrapper. Similarly, we can shut down a VM using bash external commands like "shutdown".

All the elements of a guest-based VMI can be executed faster than a hypervisor-based VMI because tracing system calls, monitoring for anomalies, and responding to anomalies do not require trapping to the hypervisor. Trapping to a hypervisor is very costly to the performance of a VM so much so that a single VM-exit can cause several microseconds or longer latency [23], depending on what is done during the VM-exit. One of the most effective ways to optimize a VMs performance is to reduce the number of VM-Exits. Traps and VM-exits are discussed in section 2.2.3, and 2.3.4, respectively.

The security advantages provided by hypervisor-based VMI systems (isolation, inspection, interposition, and deployability) are more important than those provided by guest-based VMI systems (rich abstractions and speed), becuase our fundamental priority is to maintain the goals of computer security (confidentially, integrity, authorization, availability, and accountability) throughout the life cycle of a VM. For that reason, we believe the superior security advantages of hypervisor-based VMI systems outweigh the speed and rich abstractions provided by guest-based VMI systems, even if our VMI system has a negative impact on the performance on monitored VMs (due to the need

for traping and VM-exits). As a result, we have chosen to design and implement a hypervisor-based VMI system.

1.4.3 Why use eBPF for Tracing?

As previously mentioned, an increasing number of organizations today are using cloud-computing environments and virtualization technology to run their business. In fact, Linux-based clouds are the most popular cloud environments among organizations, and thus, they have become the target of cyber attacks launched by malware [19]. As a result, security experts, and knowledgeable users are required to monitor Linux systems with the intent of maintaining the goals of computer security. The demand for protecting Linux systems has led to the creation of many tracers like perf, LTTng, SystemTap, DTrace, BPF, eBPF, ktap, strace, ftrace, and more. As a result, when designing our VMI, we had the opportunity to choose from any of the aforementioned tracers. What follows is our justification for selecting and using eBPF to trace KVM guest system calls and their corresponding guest process.

Historically, due to the kernel's privileged ability to oversee and control the entire system, it has been an ideal place to implement observability and security software. One approach that many VMI designers and developers have taken to observe a VM is to extend the capabilities of the kernel or hypervisor by modifing its source code. However, this can lead to a plethora of security concerns, as running custom code in the kernel is dangerous and error prone. For example, if you make a logical or syntaxtical error in a user space application, it could crash the corressponding user space process. Likewise, if there exists a logical or syntaxtical error in kernel space code, the entire system could crash. Finally, if you make an error in an open source hypervisor code like KVM, all the running guest VM's could crash. Recall, the purpose of a VMI system is to debug or conduct forensic analysis on a VM [20]. If the implementation hinders that purpose, it very quickly becomes an ineffective VMI system. As we will describe in subsequent chapters of this proposal, our VMI system does require a custom Linux kernel and custom KVM module. However, to limit the amount of modifications required to implement our VMI, we chose to use eBPF for two fundamental reasons. (1) eBPF applications are not permitted to modify the kernel, and (2) eBPF is a native

kernel technology that lets programs run without needing to add additional modules or modify the kernel source code.

The benefits of eBPF go well beyond just traceability. eBPF also runs with guaranteed safety with the help of a kernel space verifier that checks all submitted eBPF bytecode before its insertion into the eBPF VM [9]. For example, the eBPF verifier analyzes the program, and makes sure it conforms to a number of safety requirements, such as program termination, memory safety, and read-only access to kernel data structures [9]. For this reason, eBPF programs are far less likely to adversely impact a live production system compared to other methods of tracing (e.g. modifiing mainline Linux kernel code and/or inserting a kernel module).

Superior performance is also an advantage of eBPF, which can be attributed to several factors. On supported architectures, eBPF bytecode is compiled into machine code using a just-in-time (JIT) compiler. This saves both memory and reduces the amount of time it takes to insert an eBPF program into the Linux kernel. Additionally, speed and memory are both saved because eBPF runs in kernel space and is able to communicate with the user space via both predefined and custom Linux kernel tracepoints (see Section 2.8). As a result, the number of traps required between the user space and kernel space is greatly diminished (see Section 2.2.3).

Trust and support in eBPF has found its way into the infrastructure software layer of data centers that hold the data of millions of clients. For instance, eBPF is already being used in production at large datacenters by Facebook, Netflix, Google, and other companies to monitor server workloads for security and performance regressions [6]. Facebook has released its eBPF-based load balancer named Katran, which has now been powering Facebook data centers for several years. eBPF has also found its way into companies like Capital One and Adobe, who both leverage eBPF via the Cilium project to drive their networking, security, and observability needs in cloud environments. eBPF has even matured to the point that Google has decided to bring eBPF to its Kubernete products (GKE and Anthos) as a default networking, security, and observability software. The trust in eBPF by big companies has incentivized us, and was one of many factors into our decistion to utilize eBPF for tracing system calls and

processes.

1.4.4 Why Utilize System Calls for Introspection?

One of the questions that we considered during the design of our hypervisor-based VMI system is by asking the following question: What Linux event can be traced and monitored to identify the presence of an anomaly within a system, with a high success rate and a low false positive/negative rate? Existing research in both VMI systems like Nitro, and IPS like pH have answered the foregoing question by successfully utilizing system call as their target event. As a result, we have chosen to utilize system calls events in our VMI system. We will discuss Nitro and pH in chapter 3.

1.5 Thesis Proposal Organization

The rest of this thesis proposal proceeds as follows:

- **Chapter 2**: We comprehensively provide the background information needed for the reader to understand three things: (1) previous work related to our topic, (2) what we are attempting to accomplish, and (3) what we attempt to accomplish in the Winter 2023 term. More specifically, we will write about the characteristics of hypervisors, CPU features that are relevant to our topic, Intel VT-x, system calls, the KVM hypervisor, QEMU, VMI systems, eBPF, the Linux tracepoint API, and IDPS.
- **Chapter 3**: We will take a look at the related work that pertains to our topic. More specifically, we will write about A KVM-based hypervisor-based VMI system called Nitro. We will also write about pH, an IDPS based on system call sequences.
- **Chapter 4**: We write about the design of our VMI system.
- **Chapter 5**: We write about the implementation of our VMI system.

- **Chapter 6**: We explore our plan of action for the second term.

Background

This chapter presents the technical background information needed to understand the related work chapter, and the design of our VMI system.

- **Section 2.1**: Provides an overview of hypervisors.
- **Section 2.2**: Explains a portion of the x86-64 Intel Central Processing Unit that is relevant to our VMI system.
- **Section 2.3**: Explains the Intel Virtualization Extension.
- **Section 2.4**: Explains what a system call is from a high level.
- **Section 2.5**: Explains KVM & QEMU.
- **Section 2.6**: Briefly provides an overview of a VMI system.
- **Section 2.7**: Explains eBPF.
- **Section 2.8**: Explains The Linux kernel tracepoint API.
- **Section 2.9**: Provides an overview of what an IDPS is.

2.1 Overview of Hypervisors

As previously mentioned, a hypervisor is a software that allows virtual machines to be created and ran on a machine. Hypervisors can be divided into two types, depending on where they are located on the machine: (1) type 1 and (2) type 2.

2.1.1 Type 1 Hypervisor

Type 1 hypervisors run directly on physical hardware to create, control, and manage VMs, and do not require support from the host OS. Type 1 hypervisors are also called native or bare-metal hypervisors. The first hypervisors, which IBM developed in the 1960s were type 1 hypervisors [18]. Examples of type 1 hypervisors include, but are not limited to Xen, KVM, VMware ESX, and Microsoft Hyper-V [1].

2.1.2 Type 2 Hypervisor

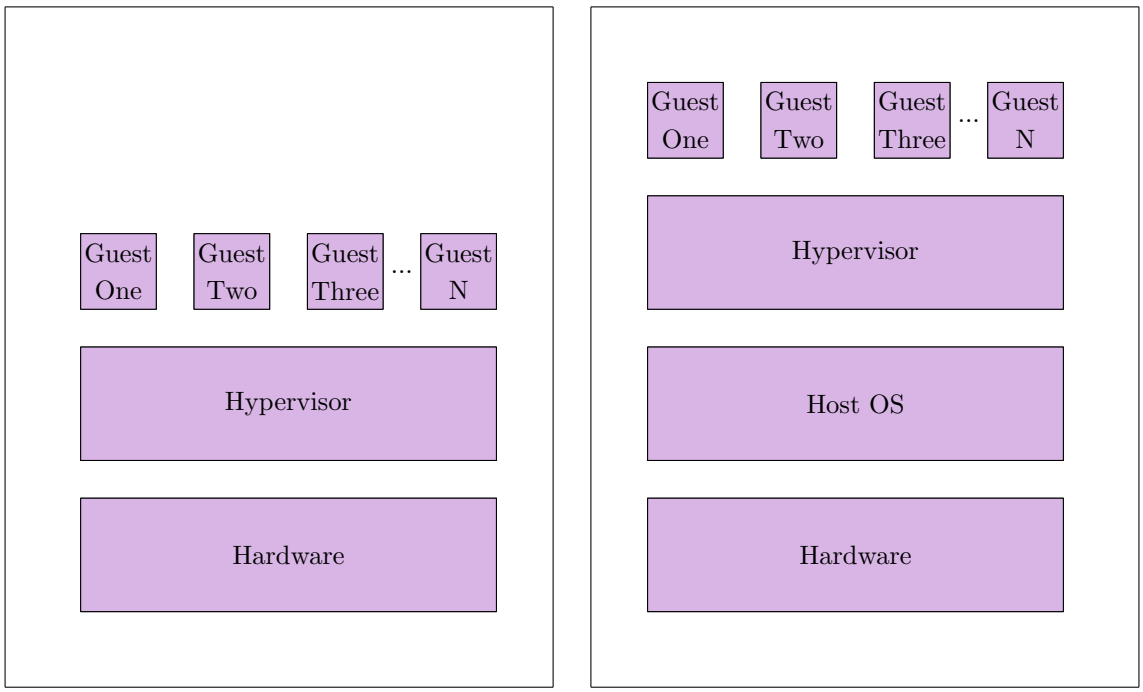
Type 2 hypervisors (also called hosted hypervisors) are installed on the OS (Windows, Linux, MacOS), similar to how computer programs are installed onto an OS. In other words, a type 2 hypervisor runs as a process on the host OS. Type 2 hypervisors abstract guest OSs from the host OS by introducing a third software layer above the hardware, as shown in figure 2.1. Examples of type 2 hypervisors include but are not limited to VMware Workstation, VirtualBox, and QEMU [1].

2.1.3 Problems With Type 1 & Type 2 Hypervisor Classifications

Although the definitions of type 1 and type 2 hypervisors are widely accepted, there are gray areas where the distinction between the two remain unclear. For instance, according to its creator RedHat, KVM is implemented and deployed using two Linux kernel modules that effectively convert the host Linux OS into a type-1 hypervisor [12]. At the same time, KVM can be categorized as a type 2 hypervisor because KVM VM's are standard Linux processes that are competing with other Linux processes for CPU time given by the Linux kernel's native CPU scheduler [15]. Due to disagreements and vagueness in the definitions of type 1 and type 2 hypervisors, two new classifications were defined: (1) native hypervisors and (2) emulation hypervisors [2].

2.1.4 Native Hypervisor

Native hypervisors are hypervisors that push VM guest instructions directly to the physical machines CPU, by using virtualization extensions like Intel VT-x (see section 2.3). [2]. Examples of Native hypervisors include but are not limited to Xen, KVM,



Type 1 Hypervisor **Type 2 Hypervisor**

Figure 2.1: Mental Model of Type 1 & Type 2 Hypervisor

VMware ESX, and Microsoft HyperV.

2.1.5 Emulation Hypervisor

Emulation hypervisors are hypervisors that emulate every VM guest instruction using software virtualization [2]. Emulated guest instructions are very easy to trace because all guest VM instructions are trapped to the hypervisor. Examples of emulation hypervisors include but are not limited to QEMU, Bochs, early versions of VMware-Workstation, and VirtualBox [2].

2.2 x86-64 Intel Central Processing Unit

2.2.1 Exceptions

Exceptions are types of signals sent from a hardware device or user space process to the

CPU, telling it to stop whatever it is doing either due to an abnormal, unprecedented, or deliberate event that occured during the execution of a program. When a user space process causes an exception, a number of steps take place. Firstly, control is transitioned from user mode (ring 3) to kernel mode (ring 0). Afterwards, CPU registers that are used by the process are saved to memory, so that the process can be loaded again in the future. The kernel will then attempt to determine the cause of the exception. Once the kernel identifies the cause of the exception, it will call the appropriate kernel space exception handler function to handle the exception. Every type of exception is assigned a unique integer called a vector [26]. When an exception occurs, the vector determines which function handler to invoke to handle the exception. If an exception is successfully handled, the CPU registers of the process that caused the exception will be restored, the process will be transitioned to user mode (ring 3), and execution will be transferd back to the user space process. It is worth noting that all the mentioned steps are dependent on the native CPU scheduler. For example, if an exception is handled successfully, the CPU may choose to resume another user space process before it resumes the one that caused the exception.

Exceptions can be divided into three categories: (1) faults, (2) traps, and (3) aborts. The goal of the background section is to solely provide information that will aid in understanding the related work, and the design and implementation of our VMI system. Faults are the only exceptions that are utilized by our VMI. Therefore, traps and aborts will not be discussed further.

2.2.2 Faults

According to standards developed by the Institute of Electrical and Electronics Engineers (IEEE), a fault is an error in a computer program's step, process, or data [8]. There exists many types of faults, which are each executed for different reasons. However, we will only introduce the Invalid Opcode (#UD) exception due to its relevance to our thesis proposal. A #UD exception, also called an undefined instruction is a fault that is generated when an instruction that is sent to a CPU is undefined (not supported) by the CPU. Some faults can be corrected (with kernel function handlers) such that the program that caused the fault may continue as if nothing happened. However, if a fault, such as a #UD exception cannot be handled successfully by a relevant kernel

function handler, then the computer will halt, and in some cases will require a reboot.

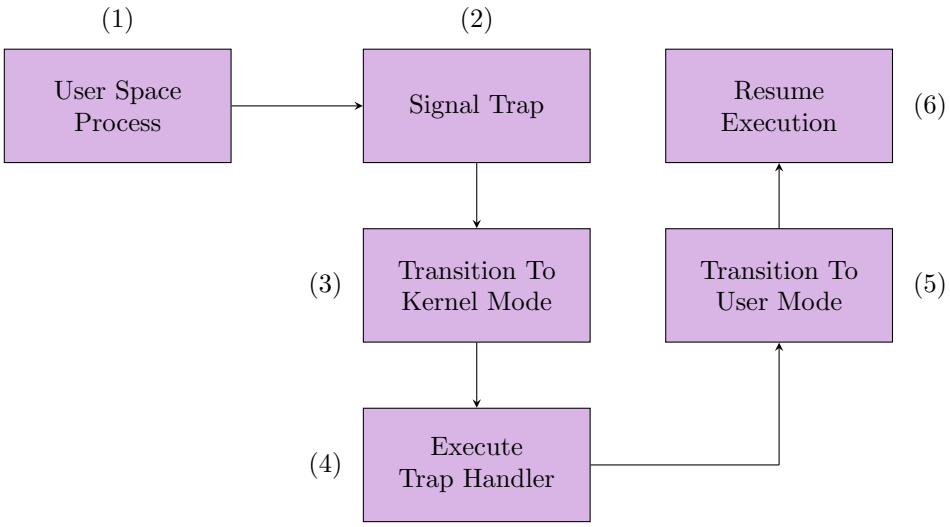


Figure 2.2: Life Cycle of an Exception

2.2.3 Instructions

In this subsection, we briefly and in a high level discuss what CPU instructions are.

An instruction is a collection of bits that instruct the CPU to perform a specific operation. According to the Combined Volume Set of Intel 64 and IA-32 Architectures Software Developer's Manual, an instruction is divided into six portions: (1) legacy prefixes, (2) opcode, (3) ModR/M, (4) SIB, (5) Displacement, and (6) Immediate. The legacy prefix is a 1-4 byte field that is used optionally, so we will not discuss it further. The opcode (2), also known as the operation code, is a 1-3 byte field that uniquely specifies and represents what operation should be performed by the CPU. Modern Intel x86_64 CPUs define many operations like SYSCALL, SYSENTER, and SYSRET, which have an opcode of 0x0F05, 0x0F34, and 0x0F07, respectively. Depending on the execution mode you are in, a user space system call request will either result in the execution of the SYSCALL, SYSENTER, or SYSRET instruction. Informally, (3), (4), (5) and (6) indicate the addresses. Addresses include operands/data that the opcode is dependent on to execute. The operands are stored in registers from which data is taken or to which data is deposited. There are two ways an operand can appear in a register. An operand can either be stored in the register (direct operand), or (2) the address of the operand can be stored in the register (indirect operand).

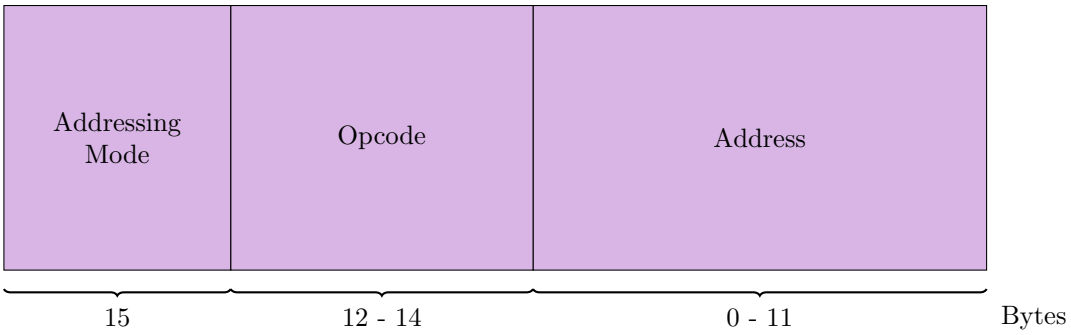


Figure 2.3: High Level Illustration of Instruction Format

2.2.4 Registers

x86-64 has 14 general-purpose registers and 4 special-purpose registers. We will only introduce the %rip, %rdi and %CR3 registers.

2.2.5 %rip Regster

%rip is a special register that holds the memory address of the next instruction to execute. %rip is an example of an indirect operand.

2.2.6 %rdi Regster

When we call a function that holds arguments, registers must be used to hold the hold the value of these arguments. %rdi is a general-purpose register that holds the data of the first argument given to a function. For example, if you called the execve system call, then %rdi would point to the filename.

2.2.7 %CR3 Register & Page Table Management

In Linux, each process has its own page table set in the kernel. Having a separate page table for each process is necessary for process isolation as they should not be allowed to access the memory of other processess. In Linux systems, there are 5 levels of page

tables: (1) Page Global Directory (PGD), Page Four Directory (PFD), Page Upper Directory (PUD), Page Middle Directory (PMD), and the Page Table Entry directory (PTE). Each process has an associated kernel-based struct mm_struct which describes its general memory state, including a pointer to its PGD page, which gives us the means to traverse the five page tables (starting with PGD). On x86-64, each time the kernel's CPU scheduler switches to a process, the process's PGD is written to the CR3 register. Thus, similar to how each process has its own kernel-based mm_struct, the value written to a CPUs CR3 register will be unique per process.

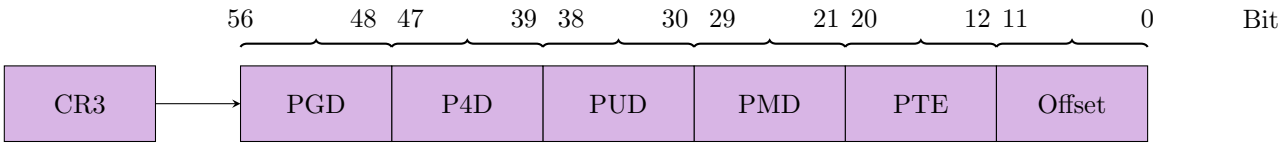


Figure 2.4: High Level Illustration of Five-Level Paging With Relation to CR3 Register

2.2.8 Protection Rings

Before we explore the hypervisor further, we must introduce protection rings (also known as privlige modes, but not to be confused with CPU modes), which is a mechanism that Intel CPUs implement to aid in fault protection. Prior to the implementation of protection rings, all the elements of a process executed in the same space. This arrangement meant that when any process generated a fault, it had the ability to affect other processes that were running normally. For example, a process that generated a fault would crash, but would also cause a perfectly running process to crash with it [28]. Due to these problems, protections rings were introduced to provide the OS with a hierarchical layer for protecting the integrity and availability of both user space and kernel space processes. With protection rings, an OSs kernel can deal with faults by terminating only the process that caused the fault.

By creating a conceptual model for protection rings, one can better understand them. Therefore, we descibe protection rings as a hierarchical system that consists of four layers: Ring 0, Ring 1, Ring 2, and Ring 3, as illustrated in figure 2.5. Next, we describe how portions of the OS are separated into each of these four rings.

First, the OS, its processes, and other components, are appointed to a specific ring. The ring that a component is appointed to, is the only place where they are permitted to execute. For instance, if process A is appointed to ring 3, and requires assistance from a ring 0 process named B, then it must conform to the following directive: each ring layer can only communicate with the layer above/below it. As an example, Ring 3 can only communicate with Ring 2, and Ring 2 can communicate with Ring 1 and Ring 3, but not Ring 0.

Ring 0 is where the OS kernel resides and runs in. Thus, it has the highest level of privilege. The kernel resides in ring 0 because it is responsible for providing services to other parts of the OS. The level of permission ring 0 provides is referred to as kernel mode, privileged mode, and/or supervisor mode. In this mode, privileged instructions are executed, and protected areas of memory can be accessed [28].

Ring 1 and Ring 2 were intended for miscellaneous non-kernel components to reside in. For example, before processsor specific virtualization extensions (like VT-x) existed, guest VMs executed in ring 1 and ring 2 [7]. However, with the advent of these virtualization extensions, guest VMs were no longer being executed in ring 1 and 2, effectively leaving them unused.

Ring 3 (also known as user mode) is where user applications and programs run. This ring has the least amount of privileges. In ring 3, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must use the Linux kernel API to access hardware or memory. As such, when certain user space process require resources from more privliged rings, the user application will issue a system call to an adjacent ring to obtain the appropriate service.

The segmentation that protection rings create, allows for process isolation, and helps ensure that one process does not adversely affect another. For example, if one process crashes due to a fault, protection rings prevents another unrelated process from crashing.

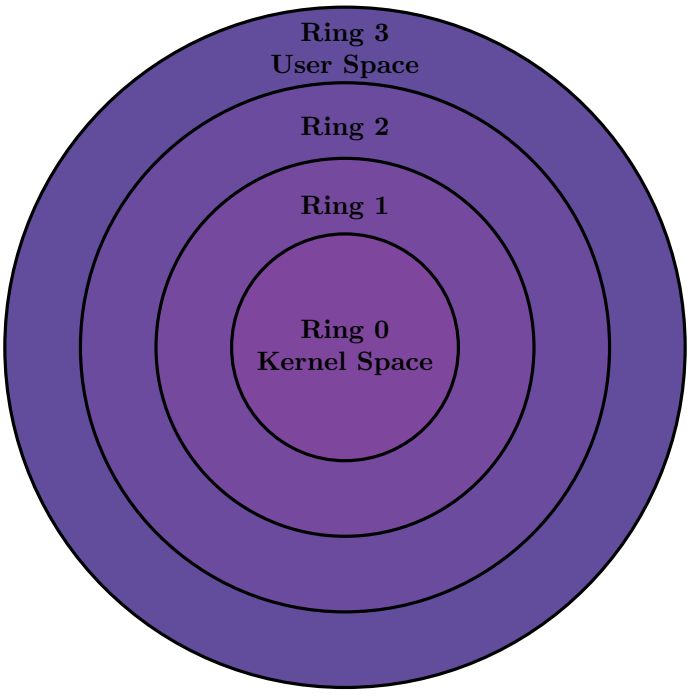


Figure 2.5: Illustration of the Intel x86 Protection Ring

2.2.9 Execution Modes

Intel x86 processors have been able to remain mostly backwards compatible due to their philosophy of extending and preserving relevant features rather than discontinuing them. An example is the extension and preservation of their execution modes. At first, there was only was one execution mode called "real mode". Real Mode is a 16 bit mode that is now present on all x86 processors. It was the first x86 mode design and was used by many early operating systems. The mode gets its name from the fact that addresses in real mode always correspond to real locations in memory instead of virtual addresses. Real mode does not support the ring protection we described in section 2.2.8. Thus, applications were able to access any region of memory. 32 bit protected mode was the successor of 16 bit real mode. It allowed the system to work with virtual address spaces, and with protection mechanisms like rings. The SYSCALL/SYSRET pair of instructions are not compatiable on 32 bit protected mode. In contrast, the SYSENTER/SYSEXIT pair are compatiable. The successor to 32-bit protected mode is 64-bit long mode, which extends registers to 64 bits and introduces

eight new registers (r8, r9, ..., r15). More importantly, the SYSCALL/SYSRET and SYSENTER/SYSEXIT pairs of instructions are all compatiable in 64-bit long mode.

Table 2.1: System Call Instruction Compatibility Based on Execution Modes

	32-bit Protected Mode	64-bit Long Mode
SYSCALL/SYSRET	NO	YES
SYSENTER/SYSEXIT	YES	YES

2.2.10 Model Specific Register (MSR)

A model specific register (not to be confused with machine state register) is a control register first introduced by Intel for testing new experimental CPU features. For example, during the time of Intel 32 bit x86 (i386) CPUs, Intel implemented two model specific registers (TR6 and TR7) for testing translation look-aside buffer (TLB), which is memory cache used for speeding up the process of converting virtual memory to physical memory. Intel warned that these control registers were unique to the design of i386 CPUs, and may not be present in future processors. However, the TR6 and TR7 control registers were kept in the subsequent i486 CPUs. However, by the time i586 ("Pentium") was released, the TR6 and TR7 MSRs were removed. As a result, software that was dependent on these control registers would no longer be able to execute on Intel Pentium series CPUs. At first, there were only about a dozen of these MSRs, but today, there are well over 200. Some MSRs have proven to be useful, due to their proven usabililty for debugging, tracing, computer performance monitoring, and toggling of certain CPU features [14]. As a result, the Intel manual states that many of these useful MSRs are carried over from one generation processors to the next. A subset of MSRs are now deemed as permanent fixtures of the x86 architecture. For historical reasons, MSRs that are given permanent status are given the prefix "IA32_". One such MSR is the IA32 Extended Feature Enable Register (EFER).

Each MSR is a 64-bit wide data structure and can be uniqely identified by a 32-bit integer. For example, the IA32_EFER MSR can be uniqely identified by the 32-bit hexadecimal value 0xC0000080. It is possible for a subset of the 64-bit wide MSR data structure to be reserved, such that it cannot be modified by a user. However,

non-reserved bits can be set or unset by using Intel's provided WRMSR instruction. Finally, any bit (reserved or non-reserved) of an MSR can be read by Intel's provided RDMSR instruction. Each MSR that is accessed by the RDMSR and WRMSR group of instructions must be accessed by using the 32-bit unique integer identifier. The table below (Table 2.1) provides information about each of the bits of the IA32_EFER MSR data structure. It is worth mentioning that the SCE label (bit 0) is by far the most interesting bit of this particular MSR. This is because bit 0 has the capability to enable or disable the SYSCALL instruction, and its counterpart SYSRET. This bit is also interesting because it is unreserved. Thus, it can be modified by any user who has privileges to exeucte the WRMSR instruction. For example, if bit 0 is set to a value of 0, then both the SYSCALL and SYSRET instructions will be undefined by the CPU. Therefore, every attempt to execute the SYSCALL or SYSRET instruction by a machine will result in an invalid OPCODE (#UD) exception (as previously disuccsed in section 2.2.1). In contrast, if bit 0 is set to 1, then both the SYSCALL and SYSRET instructions will be defined by the CPU, and will not result in a #UD Exception. By default (unless manipulated by a user), bit 0 of the IA32_EFER MSR has bit 0 set to 1. That is why system calls are able to execute perfectly fine on our machines when 64-bit long mode is enabled.

To understand our thesis proposal, it is important to mention that according to Chapter 35 of Volume 3 of the Intel Architectures SW Developer's Guide, each type of MSR can have one of three types of scopes: (1) thread, (2) core, or (3) package. Scopes in this regard can be thought of as the region in which an MSR is visibile from. MSRs with a scope of "thread" are separate for each logical processor, and can only be accessed or modified by the specific logical processor that it is assigned to [14]. MSRs with a scope of "core" are separate for each core, so they can be accessed by any logical processor (thread context) running on that core [14]. Lastly, MSRs with a scope of "package" are global, so access from any core or thread context in that will affect the entire CPU [14].

Table 2.2: IA32_EFER MSR (0xC0000080)

Bits(s)	Label	Description
0	SCE	System Call Extensions
1-7	0	Reserved
8	LME	Long Mode Enable
9	0	Reserved
10	LMA	Long Mode Active
11	NXE	No-Execute Enable
12	SVME	Secure Virtual Machine Enable
13	LMSLE	Long Mode Segment Limit Enable
14	FFXSR	Fast FXSAVE/FXRSTOR
15	TCE	Translation Cache Extension
16-63	0	Reserved

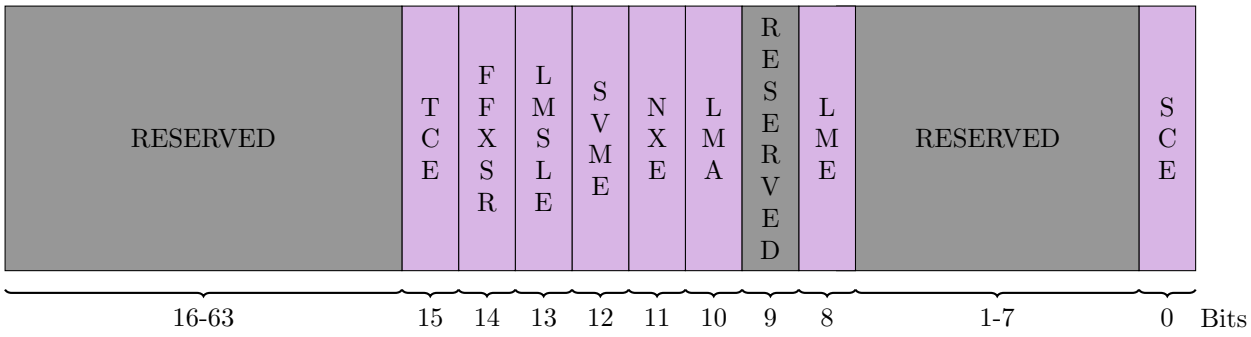


Figure 2.6: Representation of the IA32_EFER MSR (0xC0000080)

2.2.11 Supervisor Mode Access Prevention

Programmers tend to put a lot of thought into how the kernel can be protected from user space processes. The security of the system as a whole depends on that protection. For example, page tables provide a supervisor mode flag, and if set, user space is prevented from accessing it. Before the Meltdown security vulnerability [16], the kernel's memory was always mapped, so user space code could conceivably read and modify it. However, the page table supervisor mode protection flag prevented access such that any attempt by user space to examine or modify the kernel's part of the address space resulted in a segmentation violation (SIGSEGV) signal. After the Meltdown vulnerability was discovered, increased isolation was implemented with kernel page table isolation (KPTI). With KPTI, the kernel is never mapped into user space by default. Thus, just like protecting kernel space from user space, there is also value in protecting user space from the kernel. For instance, without protecting user space from kernel space, the kernel space has read and write access to user space memory mappings. This type of accessibility can lead to the development of several security exploits, including privilege escalation, which can operate by causing the kernel to access user space memory when it did not intend to.

For this reason, Supervisor Mode Access Prevention (SMAP) was introduced to Intel CPUs, and was supported by Linux beginning in kernel version 3.7. SMAP support was also added to KVM in 2014. Both host and guest OSs have SMAP enabled by default

for processors which support the feature. Intel's SMAP implementation defined a new SMAP bit in the CR4 control register (bit 21). When that bit is set, any attempt to access user space memory while running in kernel mode will lead to a fault. Of course, there are times when the kernel needs to access user space memory. In these instances, Intel has defined a separate "AC" flag that controls the SMAP feature. If the AC flag is set, SMAP protection is in force. Otherwise access to user space memory is allowed. Two new instructions (STAC and CLAC) are provided to manipulate that flag relatively quickly. STAC is used to enable SMAP, and CLAC is used to disable SMAP. For example, when the kernel space wants to access user space data using the Linux kernel API functions like get_user() or copy_from_user(), then SMAP is disabled using CLAC.

2.3 Intel Virtualization Extension (VT-X)

Intel Virtualization Extension (VT-X), also known as Intel VMX (Virtual Machine Extensions) is a set of CPU extensions that drives modern virtualization applications like KVM on Intel CPUs. Intel VT-x was released on November 13, 2005 on two models of Pentium 4 (Model 662 and 672) as the first Intel processors to support VT-x [24]. As of 2015, almost all newer server, desktop and mobile Intel processors support VT-x [24]. To maintain consistency throughout this thesis, we will only use the abbreviation VMX".

2.3.1 Overview

VMX can be viewed as a function that switches processing from a VM to the hypervisor upon detection of a sensitive instruction by the physical CPU [11]. If a guest VM is able to execute sensitive instructions on a guest system without any intervention by the host, it will cause serious problems for both the hypervisor and guest VM [11]. Therefore, it is necessary for the physical CPU to detect that the execution of a sensitive instruction is beginning and to direct the hypervisor to execute that instruction on behalf of the guest VM. However, x86 CPUs were not originally designed with the need for virtualization in mind, so there exist sensitive instructions that the CPU cannot detect when a guest VM executes them [22]. As a result, the hypervisor is unable to

execute such instructions on behalf of the guest system. Intel VT-x was developed in response to this problem [11].

Fundamentally, VMX technology introduces two new operating modes on Intel CPUs: VMX root mode and VMX non-root mode. VMX root mode is intended to be used by the hypervisor running on the host and everyting else in ring 0-3. VMX non-root mode is intended for each of the guest VMs that are powered by the hypervisor. The term "VMX root mode" is anagalous to "Ring -1", which is used to conceptualize root mode as a new layer of the protection ring as illustrated in figure 2.5. However, it is worth noting that in reality of the CPU, "ring -1" is non-existant. The Intel CPU ring privileges only consist of layers in the set {0, 1, 2, 3}. VMX root and VMX non-root mode makes use of traditional execution modes (i.e., real mode, long mode, and protected mode). As such, a VM (running in non-root mode) can make use of any of these execution modes. VMX root and non-root mode also make use of traditional protection modes. For example, both VMX root and non-root mode consists of protection ring privileges 0 to 3. The creation of VMX root and non-root mode allows the CPU and user to maintain the distinction between guest user applications and guest kernel applications automatically, essentially creating a directly comparable ring protection model (as the host OS) for each guest VM. As a result, the main purpose and motivation of introducing VMX root and non-root mode is to place limitations to the actions performed by the guest OSs, and also isolate running guest OSs from its hypervisor. Whenever a guest OS instruction tries to execute an instruction that would either violate the isolation of the hypervisor, or that must be emulated via host software, the hardware can initiate a trap, and switch to the hypervisor to handle the trap. This is very similar to the intentions of introducing a protection ring in nonvirtualized systems as explained in section 2.2.9. As a result, a guest OS can run in any privilege level without being able to impact or compromise the hypervisor hosting the VM.

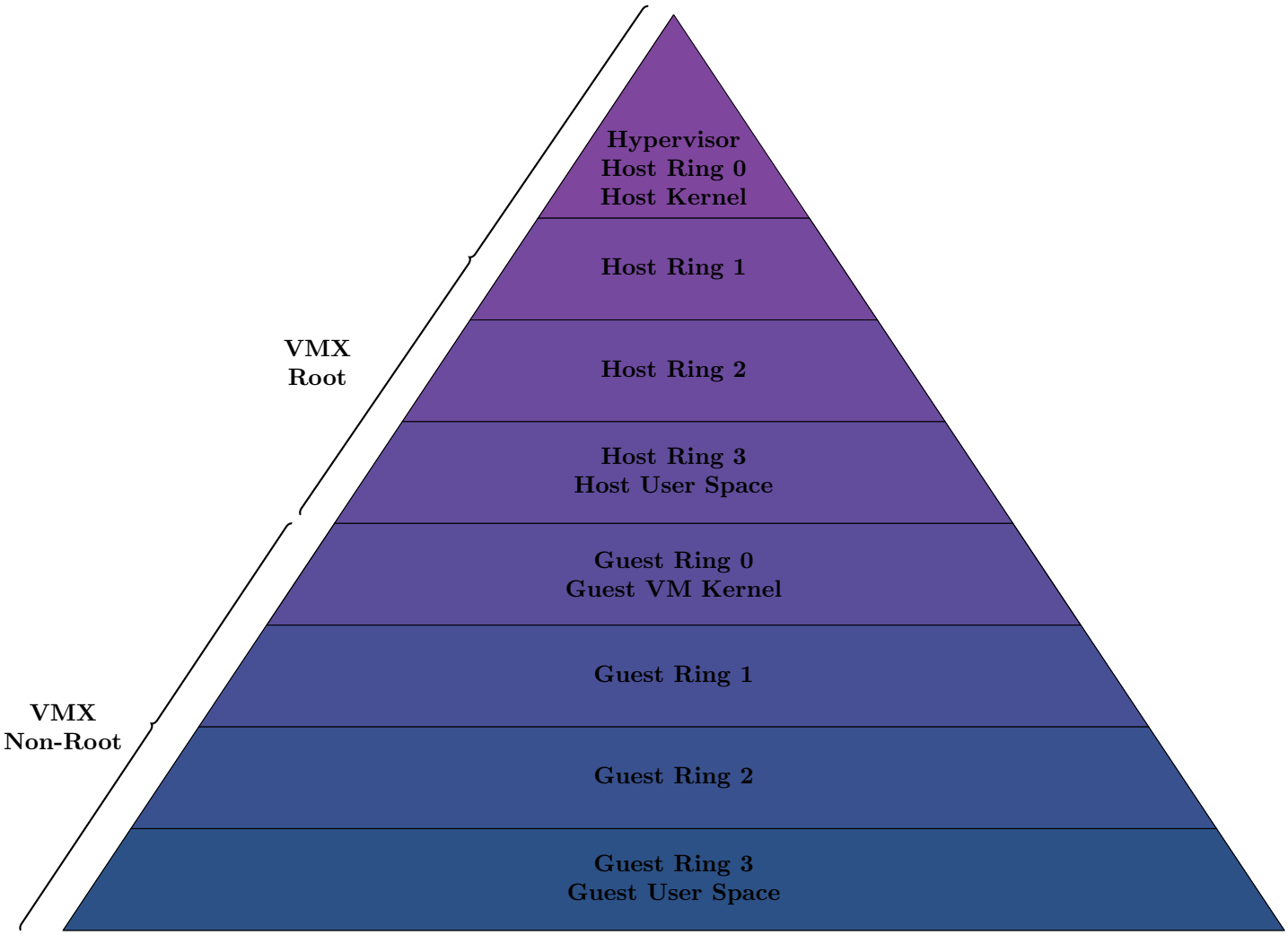


Figure 2.7: Illustration of VMX Root & Non-Root Mode in Relation to Intel Protection Rings.

2.3.2 Novel Instruction Set

VMX adds 13 new instructions, which can be used to interact and manipulate virtualization features. The 13 new instruction can be divided into three categories. Firstly, a subset of new instructions were created for interacting and manipulating the VMCS from root mode (hypervisor level). These include the VMXON, VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMLAUNCH, VMRESUME, and VMXOFF instructions. Secondly, another subset of the new instructions were created for use by the the guest VM (non-root mode). These include the VMCALL, and VM-

FUNC instructions. Lastly, there are 2 instructions that are used for manipulating translation lookaside buffer. These include the INVEPT and INVVPID instructions. Translation lookaside buffer is not relevant to this thesis. Therefore, we will not explain the INVEPT and INVVPID instructions.

VMXON

Before this instruction is executed, there is no concept of root vs non-root modes, and the physical CPU operates as if there was no virtualisation. VMXON must be executed in order to enter virtualisation. Immediately after VMXON, the CPU is placed into root mode.

VMLAUNCH

Creates an instance of a VM and enters non-root mode. We will explain what we mean by "instance of VM" in a short while, when covering VMCS. For now think of it as a particular VM created inside of KVM.

VMPTRLD

A VMCS is loaded with the VMPTRLD instruction, which loads and activates a VMCS, and requires a 64-bit memory address as it's operand in the same format as VMXON/VMCLEAR [25].

VMPTRST

Stores the current VMCS pointer into a memory address

VMCLEAR

When a pointer to an active VMCS is given as operand, the VMCS becomes non-active. [5]

VMREAD

Reads a specified field from the VMCS and stores it into a specified destination operand. [27]

VMWRITE

Writes content to a specified field in a VMCS. [28]

VMCALL

This instruction allows a guest VM (non-root mode) to make a call for service to the hypervisor. This is similar to a system call, but instead for interaction between the guest VM and hypervisor. [29]

VMRESUME

Enters non-root mode for an existing VM instance.

VMFUNC

This instruction allows the guest VM (non-root mode) to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. No VM exit occurs.

VMXOFF

This instruction is the converse of VMXON. In other words, VMXOFF exits virtualisation.

2.3.3 The Virtual Machine Control Structure (VMCS)

Additionally, a concept of the Virtual Machine Control Structure (VMCS) is introduced. The VMCS is a structure that is responsible for state-management, communication and configuration between the hypervisor and the guest VM. It contains all the information needed to manage the guest VM. A hypervisor maintains N virtual central processing units (VCPUS), where N is the product of the number of VMs running on the hypervisor and the number of VCPUs running on each VM. In other words, there exists one VMCS for each VCPU of each virtual machine. However, only one VMCS is present

on the physical processor at a time.

A VMCS can be manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE. For example, the VMPTRLD instruction is used to load the address of a VMCS, and VMPTRST is used to store this address to a specified address in memory. As there can exist many VMCS instances, but only one active one at one time, the VMPTRLD instruction is used on the address of a particular VMCS to mark it active. Then, when VMRESUME is executed, the non-root mode VM uses that active VMCS instance to know which particular VM and vCPU it is executing as. The particular VMCS remains active until the VMCLEAR instruction is executed with the address of the running VMCS. The VMCS can be accessed and modified through the new instructions VMREAD and VMWRITE. All of the new VMX instructions above require root 0, so they can only be executed from the kernel space.

More formally, a VMCS is a contiguous array of fields that is grouped into six different sections: (1) host state, (2) guest state, (3) control, (4) VM entry control, (5) VM exit control, and (6) VM-exit information.

- Host state: The state of the physical processor is loaded into this group during a VM-exit.
- Guest state: The state of the VCPU is loaded from here during a VM-entry and stored back here during a VM-exit.
- Control: Determines and specifies which instructions are allowed and which ones are not allowed during non-root mode. Instructions that are defined as not allowed, will result in a VM exit to the hypervisor (root mode);
- VM-entry control: These fields governs and defines the basic operations that should be done upon VM-entry.For example, what MSRs should be loaded on VM-entry.
- VM-exit control: VM-exit control fields governs and defines the basic operations

that must be done upon a VM-exit. For example, it defines what MSRs need to be saved upon VM-exit.

- VM-exit Information: Provides the hypervisor with additional information as to why a VM-exit took place. This field of the VMCS can be especially useful for debugging purposes.

2.3.4 VM-Exit

VM-exits is considered to be a trap that transfers control from the guest VM (non-root mode) back to the hypervisor (root mode). For a VM-exit to be successful, the given steps must take place. Firstly, the state of the running VCPU that caused the VM-exit must be saved in the "guest state" section of the VMCS. This includes information about guest MSRs. Second, information about the reason for the VM-exit must be written into the "VM-Exit Information" section of the VMCS. These should all take place before the execution is handed over to the hypervisor. When execution is given to the hypervisor, the hypervisor will handle the instruction that the guest OS could not execute by using a handler function. The handler function that is used by the hypervisor is solely dependent on the reason for the VM-exit, which is expressed in the "VM-Exit Information". For example, if a undefined instruction (#UD exception) caused a VM-exit, then the hypervisor will use the following handler function to emulate the instruction that the guest VM could not execute:

```
int handle_ud(struct kvm_vcpu *vcpu){
    static const char kvm_emulate_prefix[] = { __KVM_EMULATE_PREFIX };
    int emul_type = EMULTYPE_TRAP_UD;
    char sig~\cite{10.1007/978-3-642-25141-2_7}; /* ud2; .ascii "kvm" */
    struct x86_exception e;
    if (unlikely(!kvm_can_emulate_insn(vcpu, emul_type, NULL, 0)))
        return 1;
    if (force_emulation_prefix &&
```

```
        kvm_read_guest_virt(vcpu, kvm_get_linear_rip(vcpu),
                sig, sizeof(sig), &e) == 0 &&
        memcmp(sig, kvm_emulate_prefix, sizeof(sig)) == 0) {
        kvm_rip_write(vcpu, kvm_rip_read(vcpu) + sizeof(sig));
        emul_type = EMULTYPE_TRAP_UD_FORCED;
    }
    return kvm_emulate_instruction(vcpu, emul_type);
}
```

Listing 2.1: /arch/x86/kvm/x86.c:6959 — Linux kernel V5.18.8

Next, the changes that the hypervisor made to the state of the guest VM will be saved to the guest state section of the VMCS, so that the guest VM can continue running as if it successfully executed the instruction that caused the VM-exit. Finally, a VM-entry will occur using the VMRESUME instruction.

Certain VM-exits occur unconditionally. For example, when a VM attempts to execute an instruction that is prohibited in the guest VM (non-root mode), the VCPU immediately traps to the hypervisor (root mode). Another example of a unconditional VM-exit is if MSRs were manipulated (with the help of the Intel defined WRMSR instruction) such that an instruction was made undefined. VM-exits can also occur conditionally (e.g., based on control bits in the VMCS). For example, the hypervisor can set a bit in a specfic field of the control section of the VMCS such that whenever a VM guest VCPU encounters a RDMSR instruction, a VM-exit to the hypervisor is performed. The following is a list of instructions that could cause VM-exits in VMX non-root operation depending on the setting of the "VM-execution control" section of the VMCS:

Table 2.3: Instructions that could cause conditional VM-exits as defined by the VM-exit control section of the VMCS

Instruction
CLTS
ENCLS
HLT
IN
INS/INSB/INSW/INSD
OUT
OUTS/OUTSB/OUTSW/OUTSD
INVLPG
INVPCID
LGDT
LIDT
LLDT
LTR
SGDT
SIDT
SLDT
STR
LMSW
MONITOR
MOV from CR3/CR8
MOV to CR0/1/3/4/8
MOV DR
Continued on next page

Table 2.3 – Continued From Previous Page

Instruction
MWAIT
PAUSE
RDMSR
WRMSR
RDPMC
RDRAND
RDSEED
RDTSC
RDTSCP
RSM
VMREAD
VMWRITE
WBINVD
XRSTORS
XSAVES

Currently, there are 69 different VM-exit codes (characterized by their exit reason) specified by the Intel 64 and IA-32 Architectures Software Developer's Manual.

Table 2.4: Intel VMX Defined VM-Exits

VM-Exit Code	**Corresponding Name**
0	Exception or NMI
1	External interrupt
Continued on next page	

Table 2.4 – Continued From Previous Page

VM-Exit Code	Corresponding Name
2	Triple fault
3	INIT signal
4	Start-up IPI
5	I/O SMI
6	Other SMI
7	Interrupt window
8	NMI window
9	Task switch
10	CPUID
11	GETSEC
12	HLT
13	INVD
14	INVLPG
15	RDPMC
16	RDTSC
17	RSM
18	VMCALL
19	VMCLEAR
20	VMLAUNCH
21	VMPTRLD
22	VMPTRST
23	VMREAD
24	VMRESUME
25	VMWRITE
26	VMXOFF
27	VMXON
28	CR access
29	MOV DR
30	I/O Instruction
31	RDMSR

Continued on next page

Table 2.4 – Continued From Previous Page

VM-Exit Code	Corresponding Name
32	WRMSR
33	VM-entry failure 1
34	VM-entry failure 2
36	MWAIT
37	Monitor trap flag
39	MONITOR
40	PAUSE
41	VM-entry failure 3
43	TPR below threshold
44	APIC access
45	Virtualized EOI
46	GDTR or IDTR
47	LDTR or TR
48	EPT violation
49	EPT misconfig
50	INVEPT
51	RDTSCP
52	VMX timer expired
53	INVVPID
54	WBINVD/WBNOINVD
55	XSETBV
56	APIC write
57	RDRAND
58	INVPCID
59	VMFUNC
60	ENCLS
61	RDSEED
62	Page-mod. log full
63	XSAVES
64	XRSTORS

Continued on next page

Table 2.4 – Continued From Previous Page

VM-Exit Code	Corresponding Name
66	SPP-related event
67	UMWAIT
68	TPAUSE
69	LOADIWKEY

To synthesise all the information above about VM-exits, we will explain the cycle of a VM-exit with respect to an example in which an undefined instruction causes a VM-exit with exit code 0 (exception or NMI). As previously mentioned, an undefined instruction, also called an illegal opcode is a fault that is generated due to an instruction to a CPU that is not supported by the CPU either due to the instruction being undefined by the CPU designer, or because a user manipulated the relevant CPU MSR(s) in order to make the instruction undefined by the CPU.

For this example, we assume that virtualization is turned off. For that reason we begin by making the the physical CPU execute the VMXON instruction to start virtualisation and put itself into VMX root mode. In Figure 2.5, this is illustrated by (1). Next, the hypervisor executes a VMLAUNCH instruction in order to pass execution to the guest VM (non-root mode). We do not use the VMRESUME instruction because we are assuming that the guest VM was not previously running (as we just used the VMXON instruction to enable virtualization). In Figure 2.4, the guest VM starting is illustrated by (2). The VM instance runs its own code as if running natively until it attempts to execute an instruction that is either undefined or defined to result in a VM-exit by the control section of the VMCS. In both cases, it will result in a VM-exit. However, it is worth mentioning that in our example, the guest ran an undefined instruction and not an instruction that was governed by the VMCS to result in a VM-exit. This is illustrated in Figure 2.5 by (3). The hypervisor will consult the "VM-exit information" section of the VMCS to look into why the cause of the VM-exit. Based on the information provided by the "VM-exit Information" section of the VMCS, the hypervisor will take appropriate action by using a handler relevant to the exit reason.

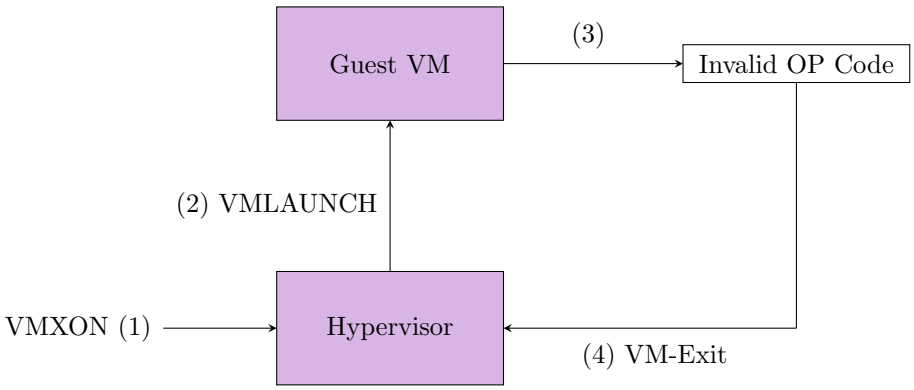


Figure 2.8: Life Cycle of a VM-Exit on invalid opcode

2.3.5 VM-Entry

VM-entry transfers control from the hypervisor (VMX root mode) back to the guest VM (VMX non-root mode). Software can enter VMX non-root operation using either of the VM-entry instructions VMLAUNCH and VMRESUME. For example, if the guest VCPU is not yet running (due to a prior VMCLEAR instruction), then it will use VMLAUNCH. In the case of a VM-exit, it will use VMRESUME [31]. Before a VM-entry can commence, the hypervisor executes dozens of checks to ensure that the state of the VMCS is correctly configured such that the subsequent VM-exit can be supported, and and the guest conforms to IA-32 and Intel 64 architectures [11].

To help understand the purpose and relevance of VM-entry within the life cycle of a hypervisor with guest VMs, we will explain the cycle of a VM-entry as illustrated in Figure 2.6. In this example, we assume that the virtualization is not enabled. Thus, we execute the VMXON instruction and enter into the hypervisor (VMX root mode). Next, we execute VMLAUNCH (VM-entry) to start the guest VM.

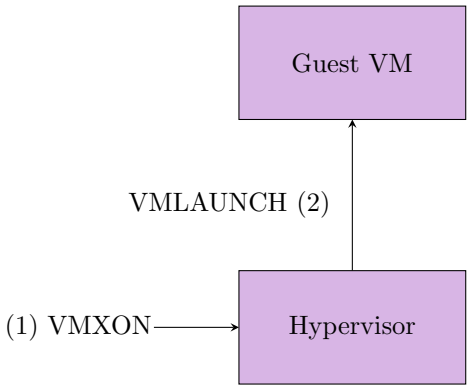


Figure 2.9: Life Cycle of a VM-Entry

Now that we have introduced the background information of VMX, we can give an overview of the life cycle of a hypervisor. First, a program executing in ring 0 needs to execute the VMXON instruction to enable virtualization and enter into VMX root mode. At this point, the program is considered a hypervisor. This is illustrated in figure 2.7 with (1). Second, the hypervisor sets up a valid VMCS with the appropriate control bits set. Third, the hypervisor can launch a VM with the VMLAUNCH (VM-Entry) instruction, which transfers execution to the VM for the first time. If the VM-Entry was successful, the hypervisor will now wait for the guest to trigger a VM-exit. If the VM-entry failed, then the VMLAUNCH instruction would return an error, and control would remain within the hypervisor. Assuming that the VM-entry succeeded, and the guest ran an instruction that was prohibited, the guest will trigger a VM-exit, causing the hypervisor to regain control. This is illuestrated by (3). Fourth, the hypervisor transfers execution control back to the VM by executing the VMRESUME instruction (4), and we effectively go back to step (3). Alternatively, the hypervisor can also stop the VM and disable VMX by executing VMXOFF, as shown by (4).

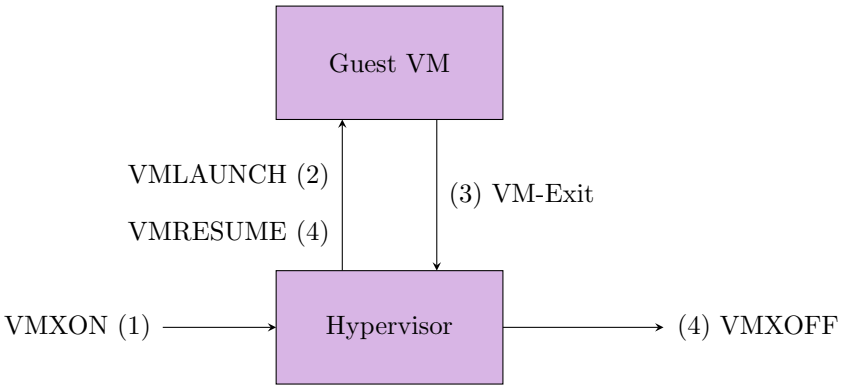


Figure 2.10: Successful Hypervisor Life Cycle Under Intel VMX

2.4 System Calls

As previously mentioned, modern computers are divided into two modes: user mode (ring 3) and root mode (ring 0). Computer application such as Microsoft Teams resides in user space (ring 3), while the underlying code that runs the operating system exists in kernel space (ring 0). By design, user space processes cannot directly interact with the kernel space. Instead, the operating system provides an API for user space processess to interact with the kernel, when it is in need of its services. This API is known as system calls. x86 CPUs define hundreds of system calls, which the operating system utilizes. Each system call has a vector that uniquely maps it. For instance, in the x86_64 architecture, the mmap system call corressponds to vector 9, and the brk system call corressponds to vector 12. The system call vector is used to find the desired kernel function for the request.

There are three types of system call instructions defined by x86 CPUs: (1) SYSCALL, (2) SYSRET, and (3) SYSENTER. The SYSCALL instruction is used when the system is in long mode.

2.5 The Kernel Virtual Machine (KVM) Hypervisor & QEMU

Kernel-based Virtual Machine (KVM) is an open-source hypervisor implemented as two Linux kernel modules The first KVM kernel module inserted into the Linux kernel is

called kvm.ko, and is architecture independent [7]. The second KVM kernel module is architecutre dependent [7]. Therefore, if the machines physical CPU is Intel based, kvm-intel.ko will be inserted into the Linux kernel. If the machines physical CPU is AMD based, then kvm-amd.ko will be inserted [7]. The insertion of the two kernel modules transforms the Linux kernel into a hypervisor. KVM was merged into the mainline open-source Linux kernel in version 2.6.20, which was released on February 5, 2007. Since its inception into the Linux kernel, Linux kernel developers have helped extend the functionality of KVM [11]. This section begins by explaining how KVM works and describes its internal and external components. KVM requires a CPU with hardware virtualization extensions, such as Intel VT-x or AMD-V. Our discussion will assume that KVM is utilizing Intel VMX virtualization extension.

KVM is structured as a Linux character device file. The kernel module creates a character device named "/dev/kvm", which can be used as an API to interact or maniplate with KVM VMs. In order to access this API, one must make use of the ioctl (input/output control) system call. The ioctl system call takes a file descriptor and a request as arguments. The file descriptor is returned to a user upon opening the character device file /dev/kvm. The KVM API provides users with dozens of ioctl requests that can be used to interact or maniplate a KVM VM. Some of the relevant ones include KVM_CREATE_VM, which creats a new guest VM, KVM_RUN, which is a wrapper to the VMLAUNCH VMX instruction, KVM_GET_MSR, which returns a value for a specific MSR, and KVM_SET_MSR, which can be used to set a value of a specific MSR. User space VM management tools like libvirt and virt manager make use of the KVM API to manage KVM VMs.

The KVM kernel module cannot, by itself, create a VM. To do so, it must use QEMU, a host user space binary called qemu-system-x86_64. As QEMU is a host user space process, it utilizes the /dev/kvm character device file API to request the KVM kernel module to execute KVM functions. For example, QEMU is used to create a VM by using the KVM_CREATE_VM ioctl call. There is one QEMU process for each guest VM. So, if there are N guest VMs running, then there will be N QEMU processes running on the host's user space. QEMU is a multi-threaded program, and one virtual CPU (VCPU) of a KVM guest VM corresponds to one QEMU thread. Therefore,

the cycles illustrated in Figure 2.9 and Figure 2.10 are performed in units of threads. QEMU threads are treated like ordinary user processes from the viewpoint of the Linux kernel. Scheduling for the thread corresponding to a virtual CPU of the guest system. Scheduling is governed by the Linux kernel scheduler in the same way as other process threads. Unlike the KVM hypervisor, QEMU is a hardware emulator, which is capable of executing CPU instructions that are both defined and undefined by the physical CPU of your machine. QEMU is useful when the physical CPU cannot handle an instruction generated by a KVM guest VM. QEMU is able to achieve hardware emulation by using Tiny Code Generator (TCG), which is a Just-In-Time (JIT) compiler that transforms a instruction written for a given processor to another one. Therefore, KVM lets a program like QEMU safely execute instructions that resulted in a VM-exit directly on the host CPU if and only if the instruction executed by the guest VM is supported by the host CPU. If the instruction executed by the guest VM (that resulted in a VM-exit) is not supported by the host CPU, then QEMU will use the TCG to translate and execute instructions if and only if TCG is enabled. If TCG is not enabled, then QEMU cannot emulate an instruction. To aid in the understanding of the life cycle of a KVM VM, we present and explain an example (Figure 2.9) to show how QEMU and KVM would handle an arbitrary instruction X that results in a VM-exit.

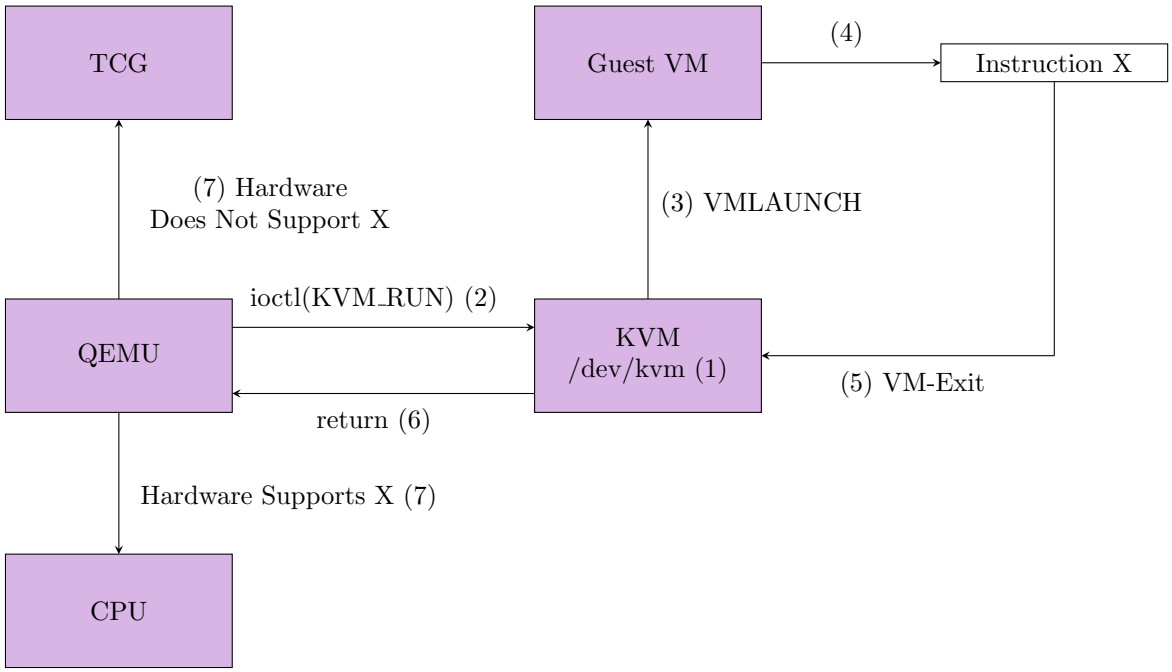


Figure 2.11: Decision on Whether QEMU use TCG or CPU for Executing an Arbitrary Instruction X.

First, a character device file named /dev/kvm is created by KVM (1). This allows QEMU to utilize this character device file to make requests to the KVM kernel module. In our case, a user requested to begin execution of a specifc guest VM. Thus, QEMU will made an ioctl() with argument KVM_RUN to instruct the KVM kernel module to start up the guest VM (2). Internally, KVM will perform a VMXON. Afterwards, KVM will will begin executing the guest VM by calling VMLAUNCH (3). The KVM guest VM will now run until it requires help from the hypervisor to execute an instruction. In our example, the guest VM attempts to execute an arbitrary instruction X (4). However, it is unable to. Therefore, a VM-exit is performed (5), and KVM identifies the reason for the exit by using the VM-exit information section of the VMCS. After the VM-exit, control is transfered to the relevant QEMU thread to decide whether the instruction X is supported by the machines CPU. If instruction X is supported by the machines CPU, then it will execute it on there (7). Otherwise, TCG will be used to emulate the instruction (7). Upon completion of the execution of instruction X, QEMU will once again makes an ioctl() system call and request the KVM to continue guest processing. In other words, the execution flow will return to step 1. This flow is repeated during the execution of a KVM guest VM until the VMXOFF instruction is executed.

We must now consider the case in which TCG was disabled either implictly (due to QEMU default settings) or explictly by the user. If TCG was disabled, then QEMU will not be able to emulate the instruction that resulted in a VM-exit, and was not capable of executing on the machine's CPU. In the case of TCG being disabled, the Linux kernel provides a number of functions that is able to emulate a non exhaustive amount of Intel x86 instructions. For example, here is the KVM function that emulates one of the three existing system call instructions provided by Intel x86.

```
static int em_syscall(struct x86_emulate_ctxt *ctxt){
    const struct x86_emulate_ops *ops = ctxt->ops;
    struct desc_struct cs, ss;
    u64 msr_data;
    u16 cs_sel, ss_sel;
    u64 efer = 0;

    /* syscall is not available in real mode */
    if (ctxt->mode == X86EMUL_MODE_REAL ||
        ctxt->mode == X86EMUL_MODE_VM86)
        return emulate_ud(ctxt);

    if (!(em_syscall_is_enabled(ctxt)))
        return emulate_ud(ctxt);

    ops->get_msr(ctxt, MSR_EFER, &efer);
    if (!(efer & EFER_SCE))
        return emulate_ud(ctxt);

    setup_syscalls_segments(&cs, &ss);
    ops->get_msr(ctxt, MSR_STAR, &msr_data);
    msr_data >>= 32;
    cs_sel = (u16)(msr_data & 0xfffc);
    ss_sel = (u16)(msr_data + 8);

    if (efer & EFER_LMA) {
```

```
        cs.d = 0;
        cs.l = 1;
    }
    ops->set_segment(ctxt, cs_sel, &cs, 0, VCPU_SREG_CS);
    ops->set_segment(ctxt, ss_sel, &ss, 0, VCPU_SREG_SS);

    *reg_write(ctxt, VCPU_REGS_RCX) = ctxt->_eip;
    if (efer & EFER_LMA) {
#ifdef CONFIG_X86_64
        *reg_write(ctxt, VCPU_REGS_R11) = ctxt->eflags;

        ops->get_msr(ctxt,
                ctxt->mode == X86EMUL_MODE_PROT64 ?
                MSR_LSTAR : MSR_CSTAR, &msr_data);
        ctxt->_eip = msr_data;

        ops->get_msr(ctxt, MSR_SYSCALL_MASK, &msr_data);
        ctxt->eflags &= ~msr_data;
        ctxt->eflags |= X86_EFLAGS_FIXED;
#endif
    } else {
        /* legacy mode */
        ops->get_msr(ctxt, MSR_STAR, &msr_data);
        ctxt->_eip = (u32)msr_data;

        ctxt->eflags &= ~(X86_EFLAGS_VM | X86_EFLAGS_IF);
    }

    ctxt->tf = (ctxt->eflags & X86_EFLAGS_TF) != 0;
    return X86EMUL_CONTINUE;
}
```

Listing 2.2: /arch/x86/kvm/emulate.c:2712 — Linux kernel V5.18.8

How does KVM know when to call em_syscall when the CPU cannot execute it, and when TCG is disabled? The answer is that KVM will fetch and decode the instruction that was provided by the guest VM by reading the "VM-exit information" section of

the VMCS. Afterwards, KVM will call an appropriate index of an opcode matrix. The index of the opcode matrix will then call em_syscall. The following snippet is a portion of the opcode matrix/table:

```
static const struct opcode twobyte_table[256] = {
    N, I(ImplicitOps | EmulateOnUD | IsBranch, em_syscall),
                                .
                                .
                                .
    N, N, N, N, N, N, N, N, N, N, N, N, N, N, N, N
};
```

Listing 2.3: /arch/x86/kvm/emulate.c:2712 — Linux kernel V5.18.8

From observing the code snippet above, we can see that if the KVM guest VM executes a syscall, and it results in a VM-exit code 0 (Exception or NMI) that cannot be handled by both the CPU and TCG, then the opcode matrix will call em_syscall and transfer execution back to the guest with a VMRESUME instruction. An example of TCG being disabled, and the SYSCALL instruction being undefined by the machines CPU is illustrated in figure 2.10.

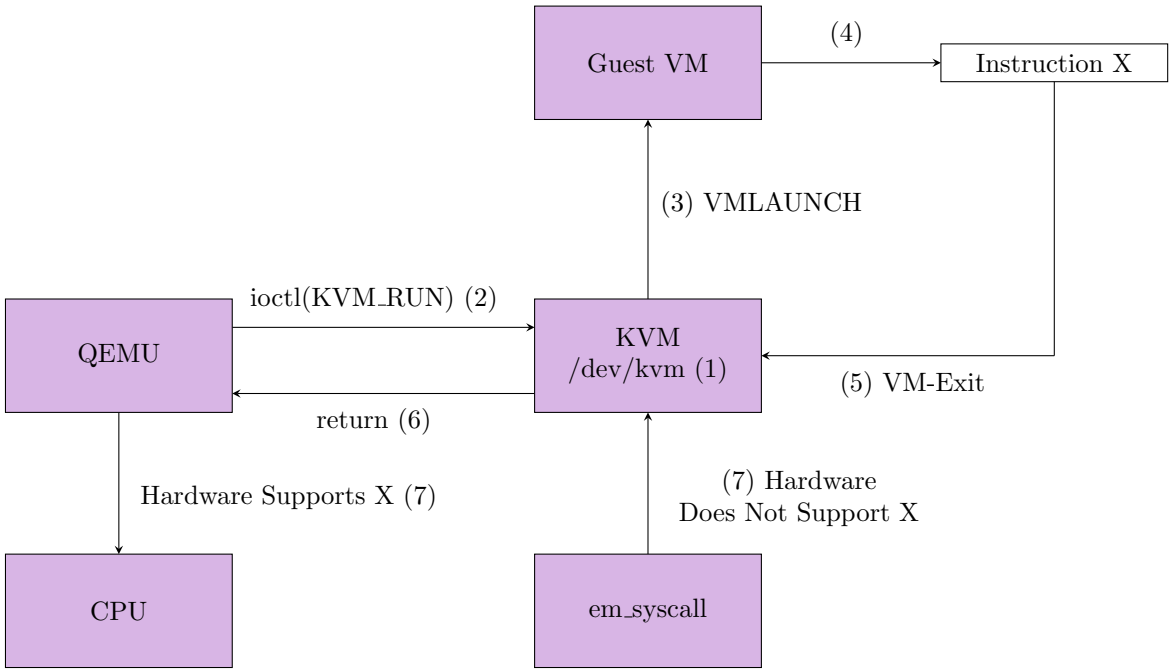


Figure 2.12: Partial KVM Life Cycle if TCG is Disabled

The worse case senario of any KVM VM is if an instruction is all three senarios are true in the event that a KVM guest VM attempted to execute an instruction that resulted in a VM-exit:

- The instruction cannot be executed on the machines physical CPU.
- The instruction cannot be executed using TCG due to it being disabled.
- THe KVM hypervisor does not support the emulation of the instruction.

If any KVM guest VM comes across this senario, then the VM will halt forever, and must be restarted.

2.6 Virtual Machine Introspection

Virtual machine introspection (VMI) is a term created by Garfinkel and Rosenblum in 2003 [10]. VMI describes the method of monitoring and analyzing the state of a virtual machine from either the hypervisor level or the guest VM all without affecting its

functionality. However, due to the existance of hypervisors, VMI is now almsot always implemented as an out-of-VM monitoring system [4]. Also, out-of-vm based monitors have been widely adopted because they run at higher privilege level and are isolated from the guest VMs that they monitor and can trap all the guest OS events as they are one layer below the guest OS. VMI allows us to take advantage of both the machine's hardware and the VMM to inspect any guest VM. The VMM is able to be manipulated ways that result important guest VM events are trapped to hypervisor. This ability to do this with a hypervisor is valuable for virtual machine introspection as it allows us to trap important actions a guest VM may execute, and inspect the guest's state at exactly that moment.

2.7 eBPF

2.7.1 Overview

eBPF is a native Linux kernel space program that allows user space programs to trace kernel space events without modifing the Linux kernel. eBPF was motivated by the need for better Linux tracing tools. It was inspired by dtrace, which is a tracing tool available for Solaris and BSD operating systems. Unlike Solaris and BSD, Linux did not have a software to provide an overview of the running systems. It was limited to specific 3rd party frameworks that utilized system calls, library calls, and kernel modules to gather infromation. Although Linux kernel modules are useful, they also pose a significant risk to the system becuase they run in kernel space. Linux kernel modules could cause the kernel to crash. In addition to having a wide range of security flaws, modules have a high overhead maintenance cost because updating the kernel could break the module. Building on the Berkeley Packet Filter (BPF), which is a software for capturing, monitoring, and filtering network traffic in the BSD kernel, a team began to extend the BPF backend to provide a similar set of features as dtrace. eBPF was first released in limited capacity in 2014 with Linux 3.18, and the full software released in Linux 4.4 and above.

2.7.2 How Does eBPF Work?

There are two ways to write eBPF programs: (1) you can inject eBPF bytecode directly into the kernel, or (2) use one of the many frontend APIs to convert a higher level language into eBPF bytecode. For example, BPF Compiler Collection (BCC) is an front end API for eBPF that allows several high level languages including Python, Go, and C++ to write eBPF programs in C to generate eBPF bytecode and submit it to the kernel. Before the bytecode is loaded into the kernel, the eBPF program must pass a certain set of requirement. This involves executing the eBPF program to a verifier to perform a series of checks. The verifier will traverse the potential paths the eBPF program may take when executed in the kernel, making sure the program does indeed run to completion without an infinite loop that would cause a kernel lockup. Other checks the verifier does include checking for valid register states, making sure the eBPF program size has a maximum of 4096 assembly instructions to guarantee that the program will terminate within a bounded amount of time, and verifies that no out of bounds memory accesses are possible. If all these checks pass, the eBPF program is then sent to a JIT compiler that translates the eBPF bytecode into the machine specific instruction set to optimize execution speed of the program. This makes eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module. Afterwards, the eBPF program is loaded into the kernel, it listens for kernel events that the eBPF program specifed to observe. Kernel events are an action or occurrence that is defined by either the Linux kernel Tracepoint API, a user defined user space event (uprobe) or a user defined kernel space event (kprobe). When these hooks are triggered, the eBPF program will capture it and transfer it to user space, allowing us to simply observe the data or manipulate it. The transfering of event data from kernel space to user space is done by a mechanism named eBPF maps. Maps can take the form of many different data structures depending on the user's needs. The ability to place hooks into almost any function with the Linux kernel Tracepoint API, uprobe, or kprobe is one of the many aspects that makes eBPF so useful. For example, a user can hook a kernel system call function, so that it can be traced every time the the system call is executed. What follows is an extensive explantion to the Linux Kernel Tracepoint API. We do not further explain uprobes or kprobes because they are not utilized in our VMI.

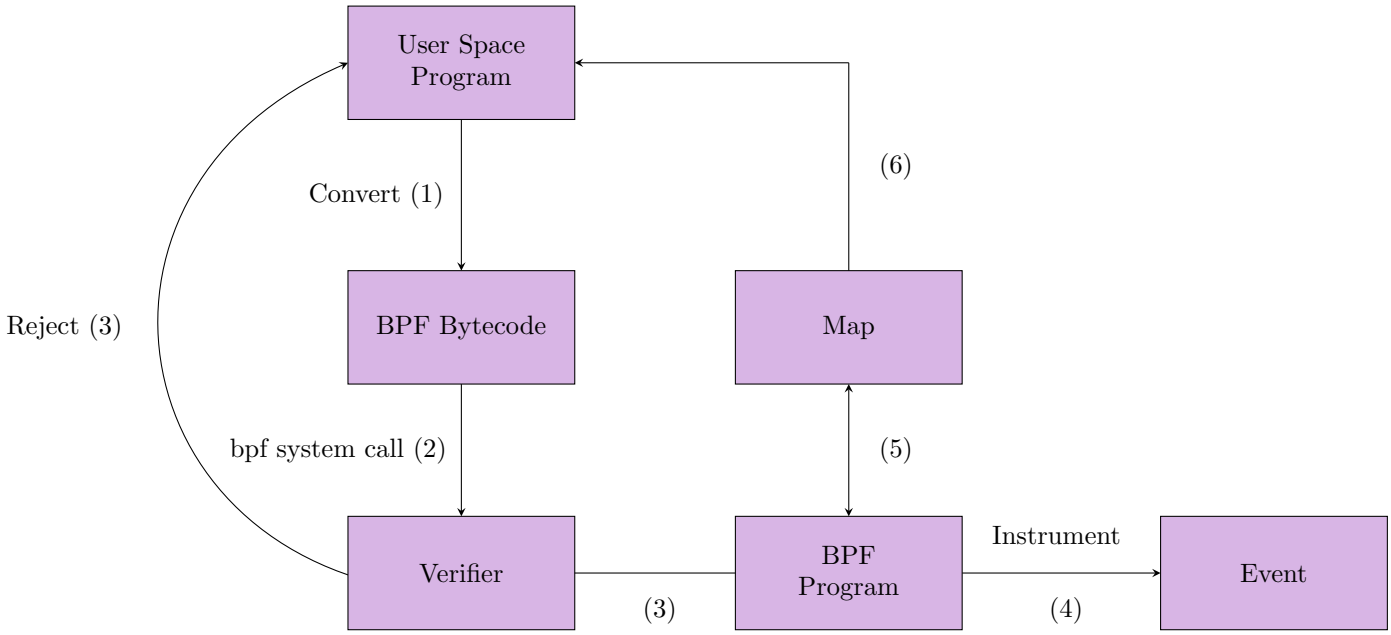


Figure 2.13: Illustration of eBPF Life Cycle

2.8 The Linux Kernel Tracepoint API

2.8.1 Overview

A Tracepoint is a marker (a piece of code) that can be hooked to certain areas of the Linux kernel source to allow for tracing kernel events at runtime and without stopping the execution of the kernel. Tracepoints are used by a number of tools for kernel debugging and performance problem diagnosis like eBPF. Although using tracepoints is ideal when possible, they have a few caveats; in particular, a limited number of tracepoints are defined by the kernel, and they do not cover an exhaustive list of kernel functionality. The offical kernel code base consists of thousands of predefined events. A small proper subset of these predefined events are KVM related. Whether that number will grow significantly is a matter of debate within the official team of Linux kernel developers community. However, as the Linux kernel is open-source, it is trivial to extend the tracepoint API to hook kernel functions in order to trace kernel events of interest.

2.8.2 Identifing Traceable Kernel Subsystems

Assuming that you have not extending the Linux kernel tracepoint API, the directories within /sys/kernel/debug/tracing/events represent the kernel subsystems that are avaiable for tracing. On Linux kernel version 5.18.8, there are 124 subsystems that are traceable by the API, which consist of the following:

Table 2.5: Traceable Kernel Subsystems

alarmtimer	gvt	kvm	mmc
clk	i2c	mmap	oom
enable	io_uring	netlink	ras
ftrace	jbd2	pwm	sched
hwmon	mei	rseq	syscalls
iomap	neigh	swiotlb	v 412
iwlwifi_msg	power	irq_vectors	xen
mce	resctrl	tlb	cfg80211
msr	sock	x86_flb	dma_fence
page_pool	thp	bpf_trace	filemap
regmap	workqueue	devfreq	header_page
skb	block	fib6	interconnect
thermal	cros_ec	bpf_trace	iwlwifi_data
vsyscall	ext4	hda_intel	mac80211
asoc	hda	intel_iommu	module
compaction	i915	irq_vectors	page_isolation
error_report	irq	kvmmmu	raw_syscalls
gpio	kmem	mmap_lock	scsi
hyperv	migrate	nmi	task
iommu	net	qdisc	vb2

iwlwifi_ucode	printk	rtc	xhci-hcd
mdio	rpm	sync_trace	cgroup
napi	spi	udp	drm
percpu	timer	xdp	fs_dax
regulator	writeback	bridge	huge_memory
smbus	bpf_test_run	devlink	iocost
thermal_power_allocator	dev	filelock	iwlwifi_io
wbt	fib	header_event	mac80211_msg
avc	hda_controller	intel-sst	mptcp
cpuhp	initcall	iwlwifi	pagemap
exceptions	irq_matrix	libata	rcu
signal	tcp	vmscan	

2.8.3 Identifing Tracepoint Events

Each subsystem consists of multuple kernel events that can be traced. For example, /sys/kernel/debug/tracing/events/kvm consists of all the KVM events that are traceable.

2.8.4 Tracepoint Format File

Each event has a format file that provides a user with arguments that can be traced from kernel space to user space eBPF programs. For example, the format file for KVM exits can be found in /sys/kernel/debug/tracing/events/kvm_exit/format, and are shown in Listing 2.4. These arguments come from the mainline kernel space KVM function __vmx_handle_exit. These arguments are passed to the kernel space tracepoint function. When tracing the kvm_exit event, the information that these arguments contain is what is stored and sent back to the user space via the eBPF map data structure. In

Listing 2.4, we can see per one exeuction of the __vmx_handle_exit function, some of the information the kvm_exit tracepoint holds is exit_reason (exit code), the rip register of that particular instruction, and the vcpu number.

```
name: kvm_exit
ID: 2059
format:
    field:unsigned short common_type; offset:0; size:2; signed:0;
    field:unsigned char common_flags; offset:2; size:1; signed:0;
    field:unsigned char common_preempt_count; offset:3; size:1; signed:0;
    field:int common_pid; offset:4;  size:4; signed:1;

    field:unsigned int exit_reason; offset:8; size:4; signed:0;
    field:unsigned long guest_rip; offset:16; size:8; signed:0;
    field:u32 isa; offset:24; size:4; signed:0;
    field:u64 info1;  offset:32; size:8; signed:0;
    field:u64 info2;  offset:40; size:8; signed:0;
    field:u32 intr_info;  offset:48; size:4; signed:0;
    field:u32 error_code; offset:52; size:4; signed:0;
    field:unsigned int vcpu_id; offset:56; size:4; signed:0;
```

Listing 2.4: Format File for the kvm_exit Linux Kernel Tracepoint Event — Linux kernel V5.18.8

2.8.5 Tracepoint Definition

Tracepoints are defined in header files under include/trace/events. Each tracepoint definition consists of a description of the following:

TP_PROTO

The TP_PROTO is the function prototype of the function that is calling the tracepoint. For example, if talking about the kvm_exit event, the TP_PROTO would be TP_PROTO(unsigned int exit_reason, unsigned long guest_rip), and would be called from the __vmx_handle_exit function for each kvm_exit a VM makes.

TP_ARGS

TP_ARGS corresponds to the parameter names, which are the same as the ones given to TP_PROTO.

TP_STRUCT_entry

TP_STRUCT_entry corresponds to the fields which are assigned when the tracepoint is triggered. For example, in the case of kvm_exits, these are the fields included in the format file in Listing 2.4 above.

TP_fast_assign

TP_fast_assign statements consist of the kernel variables that instantiate the fields found in the format file in Listing 2.4 above.

TP_printk

TP_printk is responsible for using those field values to display a relevant tracing message to programs like eBPF.

2.9 Intrusion Detecion Prevention System

2.9.1 Overview

With respect to our thesis, intrusion detection is the process of monitoring events occurring in a computer system and analyzing them for anomalies, which are violations or imminent threats of violation of our defined computer security policies. There are two types of intrusion detecetion systems: IDS and IDPS. An IDS is software that automates the intrusion detection process by implictly monitoring a computer for malicious activity. An IDPS is a software that automates the intrusion detection process, and also reponds to the malicious activity by attempting to stop it from continuing. In this

section, we will only write about IDPS. IDPS software use two notable methodologies to detect anoamlies: signature-based, and anomaly-based.

2.9.2 Signature-Based Detection

A signature is a pattern that corresponds to a known attack. Signature-based detection is the process of comparing signatures (of known attacks) against events of a computer system to identify possible attacks. Signature-based detection is very effective at detecting known attacks but largely ineffective at detecting unknown attacks. For example, if there is no signature for an arbitrary attack X, then the IDS will not be able to identify and respond to the arbitrary attack X.

2.9.3 Anomaly-Based Detection

Anomaly-based detection is the process of comparing definitions of what activity is considered normal against computer events to identify significant changes. An IDPS using anomaly-based detection has profiles that represent the normal behavior of activites for users. The profiles are developed by monitoring the characteristics of typical activity over a period of time. For example, a given profile Y for a host system might hold normal sequences of system calls of a given program K. The IDPS then uses this normal profile to compare the characteristics of future running instances of program K. If there is a strong deviation between the normal profile and the new running instance of program K, then the IDPS will detect program K as malicious and respond to it by perhaps stopping the process. Unlike signature-based detection, the major benefit of anomaly-based detection is that they can be effective at detecting unknown attacks. For example, suppose that a computer becomes infected with a new type of malware. The malware could request a large number of unique system calls that deviate from the a normal profile. Even though it is a new malware, an anomaly-based detection can still respond to it as long as it deviates from a normal profile. A disadvantage of a anomaly-based detection is that building profiles that distinguish normal behavior from anomalous behavior is very challenging because computing activity is complex and always changing. For example, if a particular program K runs a completely normal activity M that performs large amounts of system calls only once a month, the activity

may not have been observed during the training period of a normal profile. Therefore, when the normal activity M occurs, it is likely that the IDPS will consider it malicious because it significantly deviates from the existing normal profile. For that reason, program K will terminate, causing normal behavior to fall victim to the IDPS.

Related Work

3.1 Nitro

In this section, we write about Nitro, a KVM hardware-based VMI system that utilizes guest system calls for the purpose of monitoring and analyzing the state of a virtual machine. Nitro is the first VMI system that supports all three system call instructions (SYSCALL, SYSRET, SYSENTER) provided by the Intel x86 architecture, and has once been proven to work for Windows, Linux, 32-bit, and 64-bit guests. However, as previously mentioned, Nitro in its current state only works for Windows XP x64 and Windows 7 x64 due to a lack of codebase updates from the authors. What follows is an explanation of how Nitro solves the problem of tracing KVM VM system calls and processes from the hypervisor level.

3.1.1 Properties of Nitro

Nitro Mechanism for Tracing System Calls From The Hypervisor

As mentioned in our first research question, KVM is a type 1 hypervisor (see Section 2.1.1). Thus, guest instructions that are defined by the CPU (like system call instructions SYSCALL, SYSRET, and SYSENTER) are sent directly to the CPU from the VM, without requiring intervention by the hypervisor. This is a problem for hypervisor-based VMI systems because to trace VM system call events, instructions must be explictly rerouteed (trapped) and emulated at the hypervisor level instead of being sent directly to the CPU for execution. In some cases, virtualization extensions like Intel VT-x, supported trapping specific instructions that one is interested in to the hypervisor. However, during the time of Nitro's development, and even now, trapping to the hypervisor on the event of any system call instruction is not supported on Intel x86

architectures. As a result, Nitro was forced to indirectly induce a trap to the hypervisor when a system call took place. Nitro accomplished this by forcing system interrupts (e.g. page faults, general protection faults (GPF), etc) for which trapping is supported by Intel VT-x. Because Nitro is intended to work on both 32-bit and 64-bit systems, it is required to trap and emulate all three system call instructions defined by the x86 archetecture (SYSCALL, SYSRET, and SYSENTER). As they are all three system call instructions are different in their nature, a unique trapping mechanism had to be designed for each. What follows is an explanation as to how Nitro traces each type of system call instruction.

Tracing SYSCALL & SYSRET Based System Calls On 64-bit Linux systems, system calls are implemented using the SYSCALL instruction and its analogue counterpart SYSRET. Both these instructions rely on the EFERs MSR (See section 2.2.11). Thus, the SYSCALL and SYSRET instructions can effectively be turned on and off by setting and unsetting the System Call Extension (SCE) bit in the EFER MSR. A KVM VM making use of either SYSCALL or SYSRET instructions with the SCE flag unset results in an invalid opcode exception, which forces SYSCALL and SYSRET instructions to trap to the hypervisor. Once control has passed to the hypervisor, Nitro must once again differentiate between natural exceptions and those caused by Nitro's introspection. This is achieved by looking at the cause of the #UD Exception via the %rip register. If the %rip instruction does not indicate that the #UD exception was not caused by either SYSCALL or SYSRET, Nitro injects an invalid opcode exception into the guest OS and performs a VM-entry. However, if the violating instruction is, in fact, SYSCALL or SYSRET, then Nitro collects the desired information, emulates the instruction, and returns control back to the guest OS with a VM-entry.

Tracing SYSENTER & SYSEXIT Based System Calls Similar to SYSCALL and SYSRET, the SYSENTER and SYSEXIT are a pair of system call instructions that also rely on a set of MSRs, namely the SYSENTER_CS MSR, SYSENTER_ESP MSR, and SYSENTER_EIP MSR. The values in each of these MSRs are copied into specific system registers upon a call to SYSENTER. When a SYSENTER is called, Nitro saves the value of the SYSENTER_CS MSR into a local variable on the hypervisor level, and

replaces the value of the SYSENTER_CS MSR with null. This will cause each SYSENTER operation to attempt to load a null value into the CS register, thus causing a system interrupt that will cause the VM to trap to the hypervisor. Similar to tracing SYSCALL, and SYSRET instructions, Nitro must once again differentiate between natural exceptions and those caused by Nitro's introspection. This is achieved by looking at the %rip register. If the %rip instruction indicates that the instruction to execute is not either a SYSENTER or SYSEXIT, Nitro injects an invalid opcode exception into the guest OS and performs a VM-entry. However, if the violating instruction is, in fact, SYSENTER or SYSEXIT, then Nitro collects the desired information, emulates the instruction, and returns control back to the guest OS with a VM-entry.

Process Identification

The goal of our VMI is to not only trace system call instructions, but to also trace the guest processes that requested the system call. Nitro's goal was to also determine which process produced a system call, and thus implemented it into their VMI system. How does Nitro do this? They collect and store the value of the CR3 register in their database because a CR3 value can identify a process due to the fact that the CR3 value is unique to each process.

How Nitro Empowers Anomaly Detection

Nitro's implementation allows for tracing KVM guest system calls From the host. However, Nitro does not monitor for anomalous systems, nor does it respond to anamolous system calls. Instead, Nitro expects external applications to utilize Nitro's system call tracing capabilities to perform the monitoring and responding of anomalous system calls. Different applications for system call monitoring want a varying amounts of information. In some cases an application may want only a simple sequence of system call numbers, while other application may require detailed information including register values, stack-based arguments, and return values from a small subset of system calls. As Nitro cannot foresee every need of applications that conduct system call monitoring and responding, Nitro does not deliver a fixed set of data per system call. Instead, it allows the user to define flexible rules to control the data collection during system call

tracing. Based on the user specification, Nitro will then extract the system call number. It is always important to be able to determine which process produced a system call. Therefore, Nitro will also extract the process identifier. With these capabilities, Nitro can be used effectively in a variety of applications, such as machine learning approaches to malware detection, honeypot monitoring, as well as sandboxing environments.

3.2 pH (Process Homeostatis)

3.3 Contributions & Improvements On Related Work

To summarize, our contributions are as follows:

- Nitro's implementation only allows tracing of system calls of KVM VMs that are created with QEMU. Our VMI provides the ability to trace every KVM guest system call and and their corressponding guest process no matter how the KVM VM was created.
- Nitro uses Rekall, a memory forensics framework, and LibVMI to retrieve process information from KVM VMs. Rekall is now discontineud and thus is not longer maintained. Our VMI uses our own implementation to retrieve KVM guest process information by reading the %rdi register everytime an exec family system call is executed on the KVM guest VM. This way, we don't have to rely on third party software to retrieve process information.
- We extend the Linux kernel tracepoint API in the host OS to define two new events: (1) KVM guest system calls and (2) guest processess that requested a system call. The API extension allows eBPF programs to instrument these two events.
- Nitro is not capable of monitoring and responding to anomalous KVM guest system calls. With our prototype, we provide the ability to monitor and respond to anomalous KVM guest system calls by triggering the hypervisor to satisfy a variety

of security policies. More specifically, our monitoring of anomalous system calls will be done in real time with pH. And our VMI's response system will be able to effectively delay or terminate an anomalous KVM guest process. Essentially, we are including an intrusion prevention system (IPS) into our VMI.

Designing Frail

In this section, we introduce the design of our KVM and Intel VT-x exclusive hypervisor-based VMI system called *Frail*. More specifically, we discuss the design of the four notable components that make up our VMI Frail. Firstly, (1) we explain how our VMI intends to make it possible to trace every system call from any running KVM guest VM. Secondly, we explain how our VMI intends to be able to trace the guest processess that asked for services to the guest kernel by via system calls. Thirdly, we explain our design decisions on how we can integrate Somayaji's pH implementation with our VMI in order to monitor the system calls for anomalies. Lastly, we explain our design decisions as we intend to respond to anomalous system calls found by pH.

4.1 Tracing KVM VM System Calls

With virtualization support like VMX on modern CPUs, a majority of KVM guest instructions run directly on the CPU without requiring intervention by the hypervisor (see Section 2.1.4). A small proper subset of KVM guest instructions require VM-exits, and are either sent to the CPU for execution, or require emulation either by KVM or TCG (see section 2.5). By default, every system call instruction (SYSCALL, SYSRET, and SYSENTER) that is executed in a KVM VM is defined by modern Intel x86 CPUs, and do not require a VM-exit. Therefore, it runs directly from VMX non-root to the CPU (see Section 2.2.6). For this reason, it is not trivial for hypervisor-based VMI systems to trace KVM guest system calls. This is related to our first research question (see Section 1.3). What follows is our design decisions for how we successfully trace KVM guest system calls from VMX root.

4.1.1 Trapping Systen Calls from VMX Non-Root to VMX Root

Our design for tracing KVM VM system calls is based on trapping and emulating instructions. In this subsection, we will discuss our methods for trapping every system call instruction (SYSCALL, SYSENTER, and SYSRET) in the KVM guest such that it results in a VM-exit to the hypervisor (VMX root). We do this by unsetting bit 0 of the IA32_EFER MSR using the WRMSR instruction. How do we know this works? Recall from section 2.2.6 that the IA32_EFER MSR is capable of making the SYSCALL instruction undefined by the CPU if bit 0 is unset. Moreover, according to the Intel 64 and IA-32 Architectures Software Developer's Manual, when bit 0 of the IA32_EFER MSR is unset, then every SYSCALL instruction will result in a #UD exception. Recall from section 2.2.2 that a #UD exception is a fault that traps execution to root mode. In the case of a VM, a #UD exception that occurs in VMX non-root will result in a trap such that execution is transferred to VMX root, so that the #UD exception can be handled by the KVM hypervisor. In the next subsection, we explain how the trapped system call instructions are handled so that the guest VM can continue running normallly after a VM-entry.

4.1.2 Emulating SYSCALL, SYSRET, SYSENTER

Once a SYSCALL, SYSENTER, and SYSRET instruction results in a trap to VMX root due to a #UD exception, we have two choices to handle it. (1) We can utilize QEMU's TCG capabiltiies to emulate the instructions and VM-entry back into the KVM VM. (2) We can utilize KVM's emulation capabilities (see Section 2.5) to emulate the instruction and then resume execution of the VM with a VM-entry. We chose to do the latter because emulating instructions via TCG is slower than emulating via KVM's predefined emulation functions. However, one issue arises: the KVM's emulation functionality does not implement emulation for the SYSRET instruction. Thus, we have implemented our function that emualtes this instruction, which can be viewed in Figure xxx. After implementing this function, we added it to the opcode table so that every #UD exception caused by a SYSRET instruction is handled by calling our new function.

4.1.3 Ensuring Every System Call Instruction is Trapped

Recall from section 2.2.6 that MSRs with a scope of "thread" are separate for each logical processor and can only be accessed by the specific logical processor. The IA32_EFER MSR has a scope of "thread" according to the Intel 64 and IA-32 Architectures Software Developer's Manual. This means that each VCPU of a KVM VM has its own IA32_EFER MSR. For that reason, to trace every KVM guest system call of a particular KVM VM, we unset bit 0 of the IA32_EFER MSR for every VCPU that exists on the KVM VM. We do this step for every KVM VM that exists.

How do we know that a VM-exit was caused by a system call instruction, and not something else? In our design two checks are done to verify that a VM-exit was caused by a system call instruction. Recall that every #UD exception causes a VM-exit with code 0. Therefore, we filter out VM-exits to include only VM-exits with code 0. However, system call instructions are not the only instructions that result in a code 0 VM-exit. A code 0 VM-exit occurs when an NMI was delivered to the CPU. An NMI can be either a #UD exception, #BR exception, #BP exception, or #OF exception. Also, a #UD exception is not exclusively caused by an undefined system call instruction. It can be caused by any undefined instruction to a CPU. Therefore, our second check consists of checking the %rip instruction pointer. Recall that the instruction pointer %rip points to the next instruction (opcode) to execute. Therefore, we check if the first two bytes of the value that %rip points to is equal to SYSCALL (0x0F05), SYSENTER (0x0F05), or SYSRET (0x0F07). With these two checks, we can guarantee that the the instructions that we trace are only x86 defined system call instructions. This approach allows a VMI system to introspect guest system call events in the ideal way: the guest VM can stay running throughout introspection.

4.1.4 Extending Linux Kernel Tracepoint API

After trapping every KVM VM system call to VMX root, we will need a way to trace the system calls from ring 3 of VMX root. For that reason, we extend the Linux kernel tracepoint API by creating a new tracepoint that gets called whenever a KVM VM system call occurs. As eBPF programs can utilize the Linux kernel tracepoint API, we

can use it to trace these system calls from the user space. Figure 3.1 illustrates this interaction.

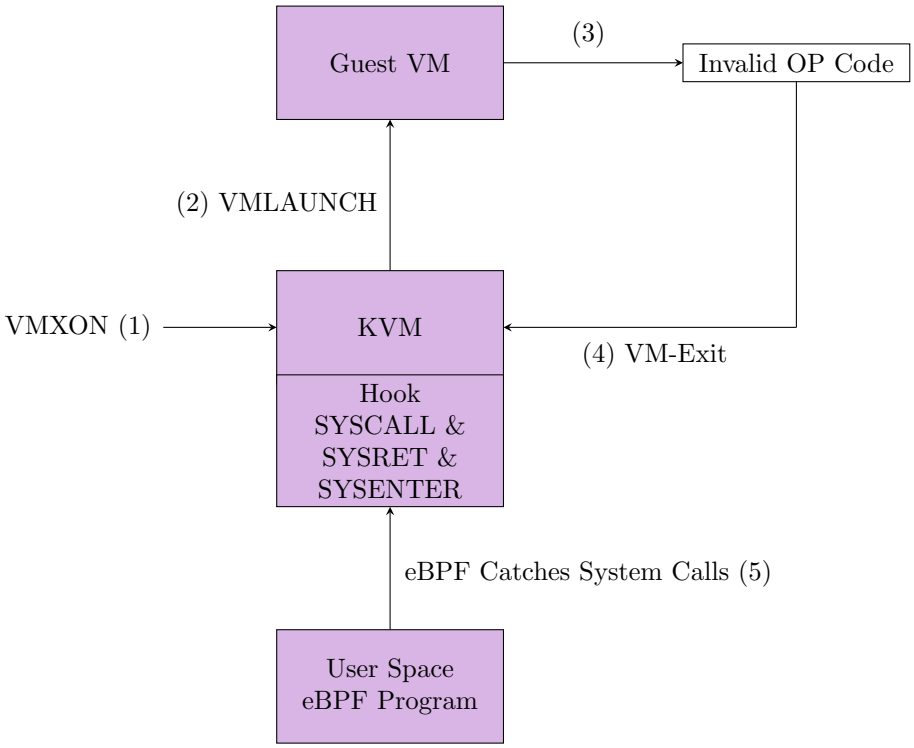


Figure 4.1: Illustration of Tracing KVM VM System Call

4.2 Tracing KVM VM Processes

Unlike VM system calls, we cannot cause a VM-exit to retrieve process information. For this reason, we must resort to a new and less trivial way to retrieve process information.

When a Linux process is executed on an x86 CPU, the CR3 register is loaded with the physical address of that process's page global directory (PGD). This is necessary so the CPU can perform translations from virtual memory address to physical memory addresses. Since every process needs its own PGD, the value in the CR3 register will be unique for each scheduled process in the system. This is very convenient for VMI because it means we don't need to constantly scan the guest kernel's memory to keep track of which process is being executed. For example, we can create a hashtable in which our keys are given by the CR3 register, and the values as a system call caused by the process corresponding to CR3. Due to the guaranteed uniqueness of the CR3 physical memory address, multiple keys will not end up with the same hashcode, thus,

a collision will never occur. The uniquenesss of CR3s help with storing processes and their corressponding system calls. However, simply tracking changes of the CR3 register doesn't give us much insight into guest processes due to the semantic gap between the VM and hypervisor. In order to bridge this gap, we need to map every address that is loaded into the CR3 register to the name of the process (the binary). In order to get the process name, we track the exec family of system calls. Why do we do this? Because to create a new proces, an exec type system call must be used. The first argument of every exec type system call requires the filename of the process. If we want to get the filename that was passed as an argument to exec, then we must read the %rdi register from the hypervisor level, which will store the virtual address of the 1st argument given to the exec function. We can then use KVM's builtin function kvm_read_guest_virt to read the virtual address given by %rdi to grant us the filename. During this process, we are accessing KVM VM user space data from the hypervisor. KVM has SMAP enabled by default, so you would think that reading VM user space data from the hypervisor would result in a fault. However, kvm_read_guest_virt uses copy_from_user to get its data. Howeber, as explained in section 2.2.11, copy_from_user temporarily disables SMAP. For of this, we are able to successfully retrieve guest VM process information directly from the KVM hypervisor.

Similar to system calls, after setting up the logic to access the KVM guest process names from the hypervisor, we need to let the host user space (VMX root ring 3) access it. Therefore, we extend the Linux kernel tracepoint API again by creating two new tracepoints. The first tracepoint will send all instances of the value that CR3 to to user space. The second tracepoint will send all instances of the value that points to %rdi to user space. Again, as eBPF programs can utilize the Linux kernel tracepoint API, we can use it to trace these process identifiers from the user space.

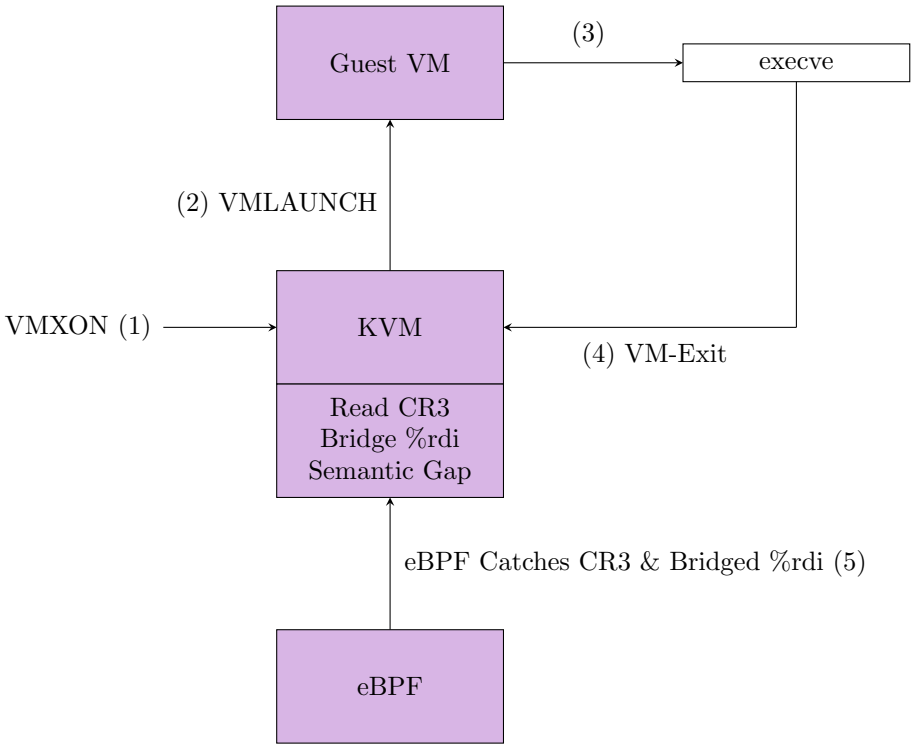


Figure 4.2: Illustration of Tracing KVM Guest Processes

4.3 Monitoring Sequence of System Calls

4.3.1 Overview

The method we present here performs anomaly intrusion detection. We build up a profile of normal behavior for a program of interest, treating deviations from this profile as anomalies. There are two stages to the anomaly detection: In the first stage we build up profiles or databases of normal behavior (this is analogous to the training phase for a learning system); in the second stage we use these databases to monitor process behavior for significant deviations from normal (analogous to the test phase). Recall that we have chosen to define normal in terms of short sequences of system calls. In the interests of simplicity, we ignore the parameters passed to the system calls, and look only at their temporal orderings. This definition of normal behavior ignores many other important aspects of process behavior, such as timing information, instruction sequences between system calls, and interactions with other processes. Certain intrusions may only be detectable by examining these other aspects of process behavior, and so we may need to consider them later. Our philosophy is to see how far we can go with

the simplest possible assumption.

4.3.2 Profiling Normal Behavior

The algorithm used to build the normal databases is extremely simple. We scan traces of system calls generated by a particular program, and build up a database of all unique sequences of a given length, k, that occurred during the trace. Each program of interest has a different database, which is specific to a particular architecture, software version and configuration, local administrative policies, and usage patterns. Once a stable database is constructed for a given program, the database can be used to monitor the ongoing behavior of the processes invoked by that program. This method is complicated by the fact that in UNIX a program can invoke more than one process. Processes are created via the fork system call or its virtual variant vfork. The essential difference between the two is that a fork creates a new process which is an instance of the same program (i.e. a copy), whereas a vfork replaces the existing process with a new one, without changing the process ID. We trace forks individually and include their traces as part of normal, but we do not yet trace virtual forks because a virtual fork executes a new program. In the future, we will switch databases dynamically to follow the virtual fork. Given the large variability in how individual systems are currently configured, patched, and used, we conjecture that individual databases will provide a unique definition of normal for most systems. We believe that such uniqueness, and the resulting diversity of

systems, is an important feature of the immune system, increasing the robustness of populations to infectious diseases [20]. The immune system of each individual is vulnerable to different pathogens, greatly limiting the spread of a disease across a population. Traditionally, computer systems have been biased towards increased uniformity because of the advantages offered, such as portability and maintainability. However, all the advantages of uniformity become potential weaknesses when errors can be exploited by an attacker. Once a method is discovered for penetrating the security of one computer, all computers with the same configuration become similarly vulnerable. The construction of the normal database is best illustrated with an example. Suppose we observe the following trace of system calls (excluding parameters):

open, read, mmap, mmap, open, read, mmap We slide a window of size k across the trace, recording each unique sequence of length k that is encountered. For example, if k = 3, then we get the unique sequences: open, read, mmap read, mmap, mmap mmap, mmap, open mmap, open, read

For efficiency, these sequences are currently stored as trees, with each tree rooted at a particular system call. The set of trees corresponding to our example is given in Figure 1.

4.4 Responding to Anomalous Behavior

When our sequences of system calls algorithm detects a malcicious process, we either terminate the malicious process, or the VM that is running the malicious process. In our user space interface, we will provide the option for users to choose from one of these two responses when the VMI detects a malicious anomaly.

4.4.1 Terminating Malicious Virtual Machine

Terminating a VM that is running a malicious process is trivial because both our VMI and the VMs are running on VMX root processes. For this reason, we can simply use the kill system call to terminate a KVM VM.

4.4.2 Terminating Malicious Process

Terminating a guest process from the hypervisor is not as trivial as terminating a VM. Like much of the complexities of our VMI, this is also due to the semantic gap. When a malicious process makes a system call, we will replace the system call with the exit system call before VM-entry to the VM. We do this by writing to the %rax register. More specifically, we replace the system call number that %rax points to with the value 60, which corressponds to the exit system call in Intel x86-64 CPUs. Recall that the exit system call is used to terminate the calling process. Due to the semantic gap, the KVM hypervisor does not have access to the PID of the process that requested the system call. However, unlike the kill system call, the exit system call does not require a pid as argument. It only requires a status code of type int as argument. Therefore, we will also have to write the %rdi register to point to an int variable. By doing this, we can effectively end any process that our sequences of system call algorithm detected as malicious. After terminating the process, the %cr3 value that coresponds to the process will be removed from our database of known KVM VM processes. An advantage to using the exit system call to terminate a process is that exit does not terminate children processes. This allows us our VMI to continue to monitor a child process of the process that was anomalous. This is advantagous because a process that is anomalous does not mean their child will also be anomalous.

Implementation of Frail VMI

5.1 User Space Component

The user space component is implemented using Python and C. The interface provides the user the option to trace KVM guest system calls from specific VCPUs based on their PID. When a user selects preferred PID, the VMI will begin to trace system calls and processes only from those VCPUs.

5.2 Kernel Space Component

The Kernel space component simply consists of a modified Linux kernel, modified KVM and extended Tracepoint API.

5.3 Extending the Linux Kernel Tracepoint API

5.4 Tracing Processess

5.5 Proof of Tracability of all KVM Guest System Calls

Future Work (Winter 2022)

6.1 Implementing Sequences of System Calls

Due to time limitations during the Fall 2022 semester, we were not able to implement the monitoring of malicious system calls using pH's implementation of sequences of system calls. We will being the next term by implementing this key feature into our VMI. This is related to our third research question as mentioned in Section 1.3.

6.2 Responding to Anomalies

6.3 Measuring Frail's Performance

After our implementation is complete, we we will conduct performance tests of KVM VM with and without our VMI enabled.

6.4 Discuss the shortcomings of our VMI System

6.5 Discuss Future Work (Beyond Winter 2022)

Bibliography

[1] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: approaches, applications, and evolutions. *ACM Computing Surveys (CSUR)*, 48(1):1–33, 2015.

[2] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Comput. Surv.*, 48(1), aug 2015.

[3] Sururah A. Bello, Lukumon O. Oyedele, Olugbenga O. Akinade, Muhammad Bilal, Juan Manuel Davila Delgado, Lukman A. Akanbi, Anuoluwapo O. Ajayi, and Hakeem A. Owolabi. Cloud computing in construction industry: Use cases, benefits and challenges. *Automation in Construction*, 122:103441, 2021.

[4] Manish Bhatt, Irfan Ahmed, and Zhiqiang Lin. Using virtual machine introspection for operating systems security education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 396–401, 2018.

[5] Ram Chandra Bhushan and Dharmendra K Yadav. Modelling and formally verifying intel vt-x: Hardware assistance for processors running virtualization platforms.

[6] Andrew Case and Golden G Richard III. Fixing a memory forensics blind spot: Linux kernel tracing. 2021.

[7] Humble Devassy Chirammal, Prasad Mukhedkar, and Anil Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[8] Nafi Diallo, Wided Ghardallou, Jules Desharnais, Marcelo Frias, Ali Jaoua, and Ali Mili. What is a fault? and why does it matter? *Innovations in Systems and Software Engineering*, 13(2):219–239, 2017.

[9] William Patrick Findlay. *A Practical, Lightweight, and Flexible Confinement Framework in eBPF*. PhD thesis, Carleton University, 2021.

[10] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206. San Diega, CA, 2003.

[11] Yasunori Goto. Kernel-based virtual machine technology. *Fujitsu Scientific and Technical Journal*, 47(3):362–368, 2011.

[12] Charles David Graziano. A performance analysis of xen and kvm hypervisors for hosting the xen worlds project. 2011.

[13] Yacine Hebbal, Sylvie Laniepce, and Jean-Marc Menaud. Virtual machine introspection: Techniques and applications. In *2015 10th international conference on availability, reliability and security*, pages 676–685. IEEE, 2015.

[14] Intel. Model specific registers and functions. In *Embedded Pentium Processor Family Developer's Manual*, Unknown.

[15] Michel Khan. Understanding kvm cpu scheduler algorithm. In *Stackoverflow*, 2018.

[16] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[17] Lan Luo, Cliff Zou, Sashan Narain, and Xinwen Fu. On teaching malware analysis on latest windows. In *Journal of The Colloquium for Information Systems Security Education*, volume 9, pages 7–7, 2022.

[18] Shannon Meier, Bill Virun, Joshua Blumert, and M Tim Jones. Ibm systems virtualization: Servers, storage, and software. *IBM Redbook, May*, 2008.

[19] Tomer Panker and Nir Nissim. Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in linux cloud environments. *Knowledge-Based Systems*, 226:107095, 2021.

[20] Bryan D. Payne. *Virtual Machine Introspection*, pages 1360–1362. Springer US, Boston, MA, 2011.

[21] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proceedings of the 1st ACM workshop on Virtual machine security*, pages 1–10, 2009.

[22] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In Tetsu Iwata and Masakatsu Nishigaki, editors, *Advances in Information and Computer Security*, pages 96–112, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[23] Wing-Chi Poon and Aloysius K Mok. Improving the latency of vmexit forwarding in recursive virtualization for the x86 architecture. In *2012 45th Hawaii International Conference on System Sciences*, pages 5604–5612. IEEE, 2012.

[24] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[25] Anil Buntwal Somayaji. *Operating system stability and security through process homeostasis*. The University of New Mexico, 2002.

[26] Unknown. Interrupt and exception handling on the x86. In *Interrupt and Exception Handling on the x86*, Unknown.

[27] Paul C Van Oorschot. *Computer Security and the Internet: Tools and Jewels from Malware to Bitcoin*. Springer, 2021.

[28] Jeffrey J. Wiley. *Protection Rings*, pages 988–990. Springer US, Boston, MA, 2011.

[29] Thu Yein Win, Huaglory Tianfield, Quentin Mair, Taimur Al Said, and Omer F Rana. Virtual machine introspection. In *Proceedings of the 7th International Conference on Security of Information and Networks*, pages 405–410, 2014.