Hypervisor-Based Intrusion Prevention System with Sequences of System Calls

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

Thank you for reading

Abstract

Hypervisors have been steadily rising in popularity, especially in the domain of cloud computing, where they play a key role in up keeping server consolidation, systems software developing and debugging, fault tolerance, system security, and workload balancing. To maintain the stability of virtualized environments, reliable techniques for virtual machine anomaly detection and prevention are required.

One of the most predominant techniques for anomaly detection and prevention in KVM virtualized environements is intrusion prevention system (IPS), a technique for monitoring the runtime state of a VM. Its usefulness has been proven by the vast number of virtual machine detection and prevention tools that have been proposed by both industry and academia. However, many existing IPS prototypes focus on guest-based deployments, which is not an ideal solution due to to guest OS tampering, and deployability issues.

For this reason, we propose a hypervisor-based intrusion prevention system, in which we discuss three key features: (1) a way to introspect guest process system calls, (2) the ability to detect and (3) prevent anomalous behavior in real-time by constructing and analyzing sequences of sytem calls of guest procesess. Additionally, we examine the limitations and challenges associated with hypervisor-based IPS using system call monitoring, such as the difficulty of accurately identifying anomalous processes, and the potential impact on system performance.

We implement a prototype for the Intel VMX component of the KVM hypervisor by integrating and improving on previous VMI systems like *Nitro*, and Dr. Somayaji's IDS named *pH*. Our evaluation shows that by manipulating each of the hypervisors

VCPUs, we are able to trace 100% of KVM guest system calls from the hypervisor via VM-exits. We also show that we are able to map each system call to its corressponding guest process by reading control registers belonging to page tables. With this, we are able to build sequences of system calls of guest procesess, and use them to detect and prevent anomalies in real-time. Finally, our benchmark tests indicate that the use of our IPS requires a greater number of VM-exits to trace guest system calls compared to when the IPS is not activated, which negatively impacts the overall system's usability, making it less efficient.

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 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 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
pH	Process Homeostasis

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 [29]. 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 [30]. 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 [30]. 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. [30]. 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 [34]. As such, a VM that is isolated may behave consistently over time, whereas one that is connected to the internet will not [30]. 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 [32]. 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) [32]. 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 [25]. 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 [30]. 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). 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) 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] [30]. 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 [27]. 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 [20]. Similarly, Windows 7 is 13 years old, and consists of only 16.06% of the same marketshare. [20].

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 [34]. 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 [26].

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 [26]. 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 [26].

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 [28], 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 [24]. 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 [25]. 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. A VMI system can be built using any specific data that has a programming interface. However, the system call interface has several unique properties that make it an excellent choice for monitoring behaviour for security violations [30].

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 IPS based on system call sequences.
- **Chapter 4**: We write about our contributions.
- **Chapter 5**: We write about the design of Frail .

- **Chapter 6**: We write about the design of Frail.
- **Chapter 7**: We discuss the shortcomings of Frail.
- **Chapter 8**: We discuss Future work.
- **Chapter 9**: This is the Appendix, which shows source code and extra information to understand my thesis.

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 IPS.
- **Section 2.3**: Explains Intel Virtualization Extension (VT-x).
- **Section 2.4**: Explains what a system call is from a high level.
- **Section 2.5**: Explains KVM & QEMU.
- **Section 2.6**: Explains eBPF.
- **Section 2.7**: Explains The Linux kernel tracepoint API.
- **Section 2.8**: Briefly provides an overview of a VMI system.
- **Section 2.9**: Provides an overview of what an VMI 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 [21]. 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 [18]. 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,

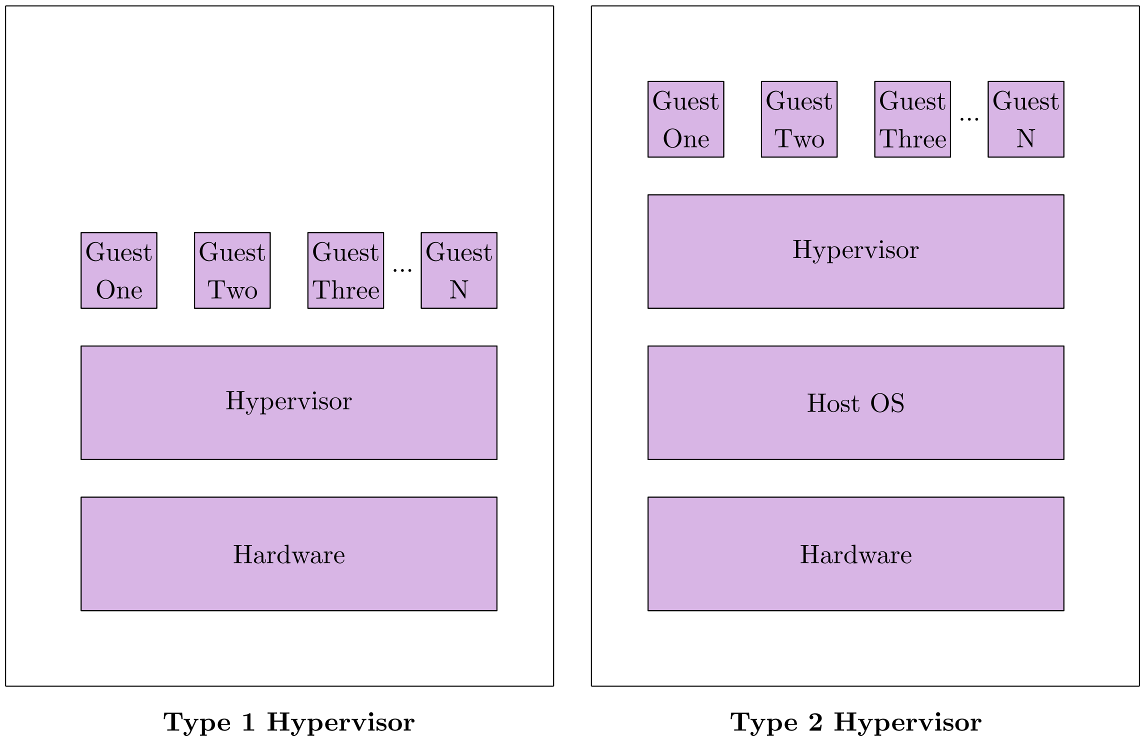

Figure 2.1: Mental Model of Type 1 & Type 2 Hypervisor

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 [31]. 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 can not be handled successfully by a relevant kernel function handler, then the computer will halt, and in some cases will require a reboot.

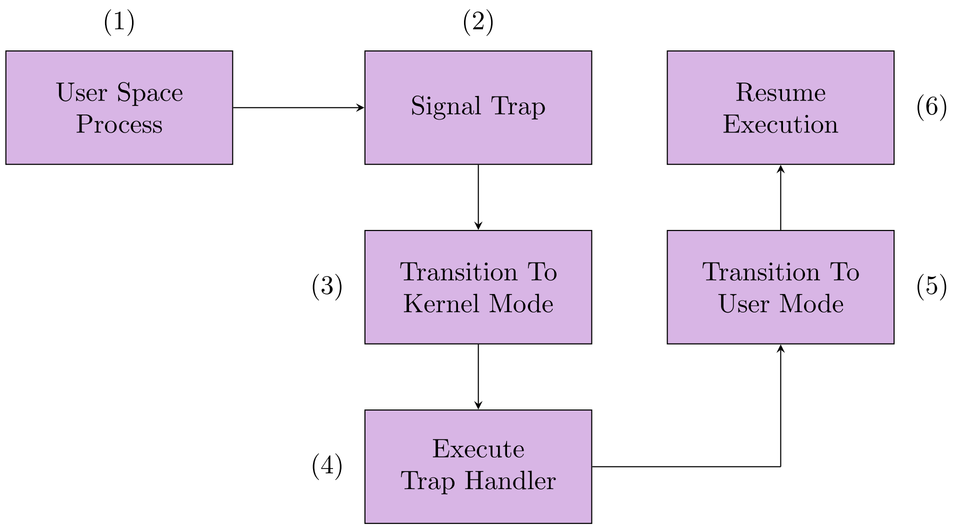

Figure 2.2: Life Cycle of an Exception

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).

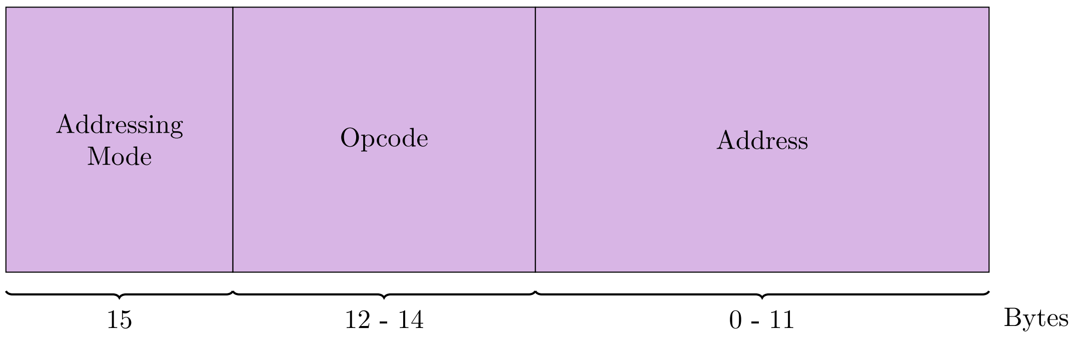

Figure 2.3: High Level Illustration of Instruction Format

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.

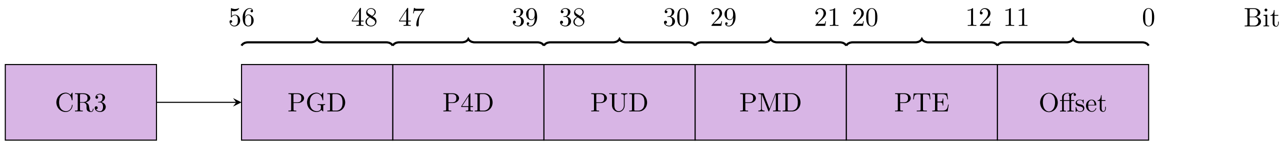

Figure 2.4: High Level Illustration of Five-Level Paging With Relation to CR3 Register

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 [33]. 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 [33].

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.

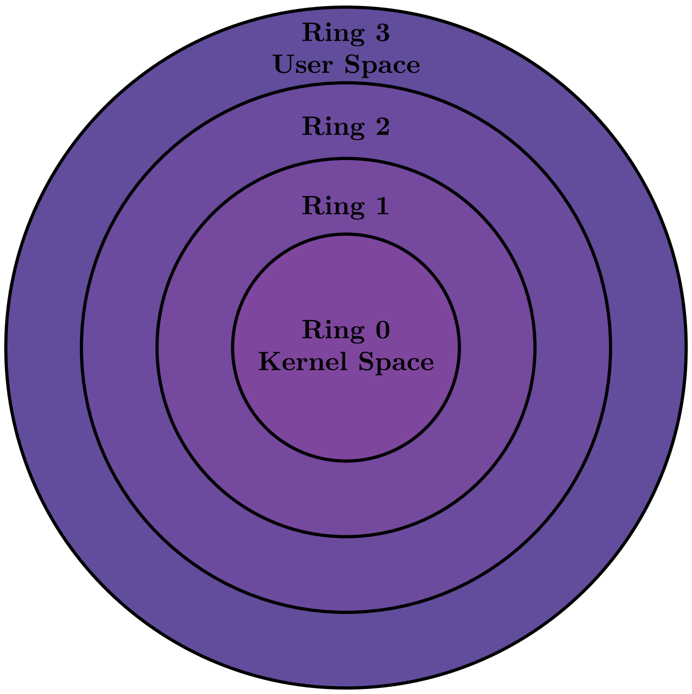

Figure 2.5: Illustration of the Intel x86 Protection Ring

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 one execution mode named ”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 of instructions 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). The SYSCALL/SYSRET pair of instructions is compatiable in 64-bit long mode. The SYSENTER/SYSEXIT pair of instructions is not compatible 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. By the time i586 ("Pentium") series CPUs were 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 [17]. As a result, the Intel manual states that many MSRs are carried over from one generation of processors to the next. A subset of MSRs are now considered 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 (IA32_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 can not 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 allows one to enable or disable the SYSCALL instructions, 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. By default (unless manipulated by a user), bit 0 of the IA32_EFER MSR is set to 1. For this reason, system calls are able to execute on our machines when 64-bit long mode is enabled.

To understand our thesis, 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 [17]. 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 [17]. Lastly, MSRs with a scope of "package" are global, so access from any core or thread context in that will affect the entire CPU [17].

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

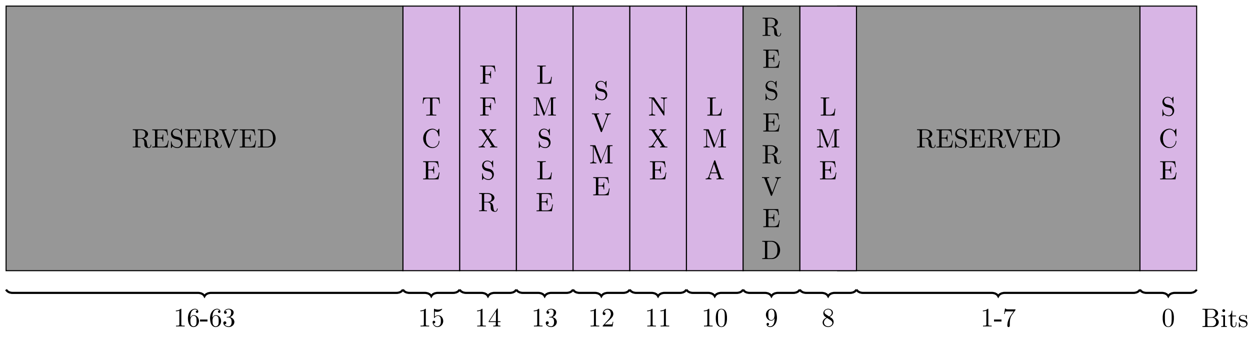

Figure 2.6: Representation of the IA32_EFER MSR (0xC0000080)

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, as a reasonable amount of the system's security depends on that protection. Page tables, for example, include a supervisor mode flag that, when set, prevents user space from accessing pages of kernel memory, even if the kernel's memory is actively mapped into user space. While the flag is set, any attempt by the user space to examine or modify pages of kernel memory results in a segmentation violation (SIGSEGV) signal. With the Meltdown security vulnerability [19] the characteristic of having kernel space memory mapped into user space allowed for unauthorized processes to read data from any address, effectively bypassing the purpose of the page tables supervisor mode flag. To mitigate the meltdown vulnerability, increased isolation was implemented with the help of kernel page table isolation (KPTI). With KPTI, kernel memory is no longer mapped into user space.

Similar to the value of protecting the kernel from user space (as mentioned above), there is also value in protecting user space from the kernel. Without doing so, the kernel space would have read and write access to mapped user space memory. 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. Due to these security exploits, Supervisor Mode Access Prevention (SMAP) was introduced to Intel CPUs, and was supported by Linux beginning in kernel version 3.7, and KVM in 2014. Both host and KVM 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 supervisor mode will lead to a page 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. The STAC instruction is used to enable SMAP, and CLAC is used to disable SMAP. A practical example of the use of SMAP is when the supervisor mode want to access user space data using the Linux kernel API functions like get_user or copy_from_user. When these Linux kernel API functions are used, SMAP is implicitly disabled by the kernel using CLAC, so that the kernel can retrieve user space memory.

2.3 Intel Virtualization Extension (VT-X)

Intel Virtualization Extension (VT-X), also known as Intel Virtual Machine Extension (VMX) is a set of CPU extensions that drives modern virtualization applications (like KVM) on modern Intel CPUs. VMX was released on November 13, 2005 on two models of Pentium 4 (Model 662 and 672) [16]. As of 2015, almost all newer server, desktop and mobile Intel processors support it [16]. To maintain consistency throughout this proposal, we will use the abbreviation ”VMX”.

2.3.1 Overview

VMX can be viewed as a function that switches execution from a VM to the hypervisor upon detecting a sensitive instruction by the physical CPU [11]. If a guest is able to execute sensitive instructions on a guest without intervention by the hypervisor, it can be problematic for both the hypervisor and guest [11]. Therefore, it is necessary for the physical CPU to detect that the execution of a sensitive guest instruction is beginning, so that the hypervisor can execute the instruction on behalf of the guest. Intel CPUs were not originally designed with VMX, so there existed cases in which the CPU could not detect when a guest executed a sensitive instruction [27]. As a result, the hypervisor was unable to execute such instructions on behalf of the guest. VMX 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, and everyting else in ring 0-3. VMX non-root mode is intended to be used by 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, protected mode, and long mode). As such, a VM can make use of any of these execution modes. VMX root and non-root mode also make use of the 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 by 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 to 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 (context switch) by switching execution to the hypervisor to handle the trap and instruction. The justification of introducing VMX root and VMX non-root is 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.

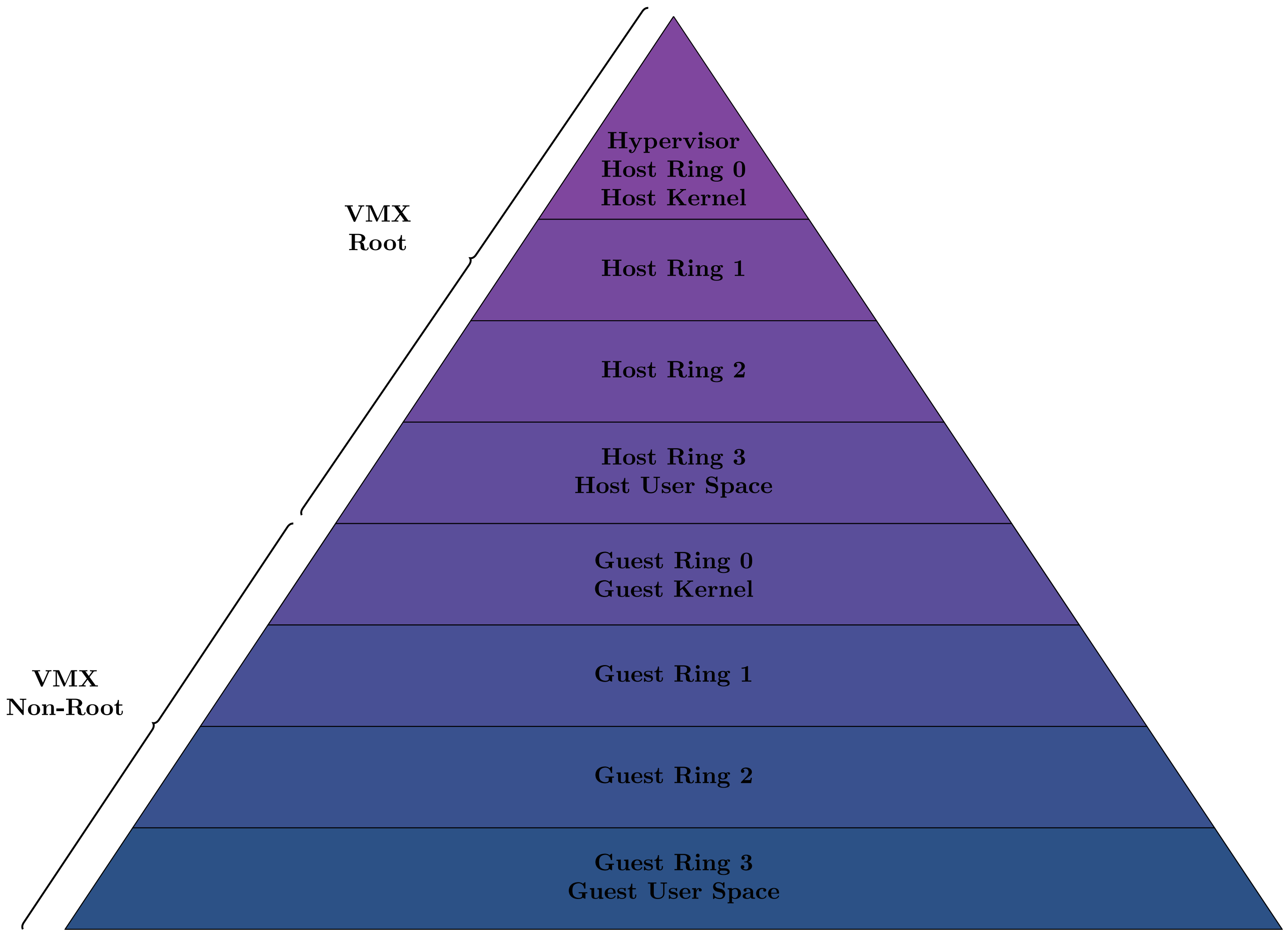

Figure 2.7: Illustration of VMX Root & Non-Root Mode in Relation to Intel Protection Rings.

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 virtualization features from VMX root mode. 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. What follows is an explanation of instructions that are relevant to our VMI system.

VMXON

Before this instruction is executed, there is no concept of root or non-root modes, and the physical CPU operates as if there is no virtualisation. VMXON must be executed in order to enable virtualization. Immediately after VMXON is executed, 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 and activated with the VMPTRLD instruction. VMPTRLD requires a 64-bit memory address of the particular VMCS that requires activation [?].

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.

VMWRITE

Writes content to a specified field in an activated VMCS.

VMCALL

This instruction allows a guest VM to make a call for service to the hypervisor. This is similar to a system call, but instead for interaction between the guest and hypervisor.

VMRESUME

Enters non-root mode for an existing VM instance.

VMXOFF

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

2.3.3 The Virtual Machine Control Structure (VMCS)

The Virtual Machine Control Structure (VMCS) is a structure that is responsible for the management, communication, and configuration between the hypervisor and guest. In other words, it contains all the information needed to manage the guest. A hypervisor maintains N number of VMCSs, where N is the product of the number of VMs running on a hypervisor and the number of VCPUs running on each VM. In other words, there exists one VMCS for each VCPU. However, only one VMCS can be active for execution on the hypervisor 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. Many VMCSs can exist, but only one can execute at a time. Thus, the VMPTRLD instruction is used on the address of a particular VMCS to mark it active.

Then, when VMRESUME is executed, the 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 by using address of the running VMCS as operand. The VMCS can be accessed and modified through the new instructions VMREAD and VMWRITE. All the new VMX instructions described above require you to be in ring 0. 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 here during a VM-entry and is saved here during a VM-exit.
- Control: Determines and specifies which instructions are allowed and which ones are not allowed while in non-root mode. Instructions that are defined as ”not allowed” will result in a VM exit.
- VM-entry control: These fields govern and define the basic operations that should be done upon VM-entry. For example, it specifies what MSRs should be loaded on VM-entry
- VM-exit control: The VM-exit control fields govern and define the basic operations that must be done upon a VM-exit. For example, it defines what MSRs need to be saved upon a 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 are 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. Secondly, 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 an appropriate handler function. The handler function that is used by the hypervisor is dependent on the reason for the VM-exit, which is expressed in the "VM-Exit Information" section of the VMCS. For example, if an undefined instruction (#UD exception) caused a VM-exit, then the hypervisor will use the VMX implemented handle_ud function handler (see Listing 2.1).

```
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 can continue running as if it successfully executed the instruction that caused the VM-exit. Finally, a VM-entry will occur with the help of the VMRESUME instruction.

Certain VM-exits occur unconditionally. For example, when a VM attempts to execute an instruction that is prohibited in the guest, the VCPU immediately traps to the hypervisor. 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 the ”control” section of the VMCS, such that whenever a VM guest VCPU encounters a RDMSR instruction, a VM-exit is performed. The following is a list of instructions that could cause conditional VM-exits depending on the configuration of the ”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

Continued on next page

Table 2.3 – Continued From Previous Page

Instruction
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
MWAIT
PAUSE
RDMSR
WRMSR
RDPMC
RDRAND
RDSEED
RDTSC
RDTSCP

Continued on next page

Table 2.3 – Continued From Previous Page

Instruction
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
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

Continued on next page

Table 2.4 – Continued From Previous Page

VM-Exit Code	**Corresponding Name**
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
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

Continued on next page

Table 2.4 – Continued From Previous Page

VM-Exit Code	**Corresponding Name**
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
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 it being undefined by the CPU designer, due to VMCS configurations, or because a user manipulated an MSR to make the instruction undefined by the CPU.

For this example, we assume that virtualization is turned off. For that reason we begin by executing the VMXON instruction to put the system in VMX root mode. In Figure 2.8, 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 VM-RESUME instruction because we are assuming that the guest VM was not previously running (as we just used the VMXON instruction to enable virtualization). This is illustrated by (2) in Figure 2.8. The VM instance runs normally until it attempts to execute an instruction that is undefined, thus it requires a VM-exit. This is illustrated 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 calling a function handler relevant to the exit reason.

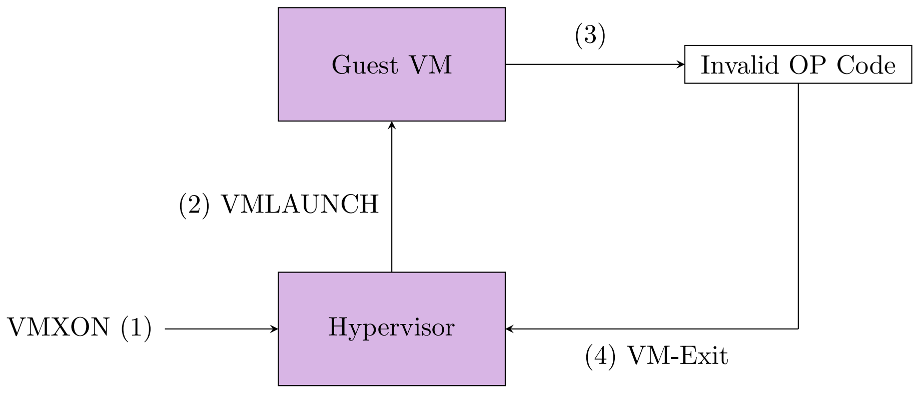

Figure 2.8: Life Cycle of a VM-Exit on invalid opcode

2.3.5 VM-Entry

VM-entry transfers control from the hypervisor (VMX root mode) back to the guest (VMX non-root mode). Software can enter VMX non-root with either the VMLAUNCH or VMRESUME instructions depending on specific conditions. For example, if the guest VCPU is not yet running (due to a prior VMCLEAR instruction call), then it will call

VMLAUNCH. However, in the case of a prior VM-exit, it VMRESUME will be called. 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 a subsequent VM-exit can be supported, and the guest conforms to IA-32 and Intel 64 architectures [11].

To help understand the purpose and relevance of a 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 2.9. In this example, we assume that the virtualization is not enabled. Thus, we execute the VMXON instruction and enter VMX root mode. Next, we execute VMLAUNCH (VM-entry) to start the guest VM.

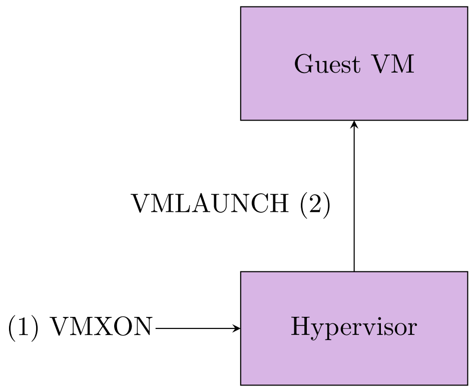

Figure 2.9: Life Cycle of a VM-Entry

Now that we have introduced the background information of VMX, we can give an overview of the complete 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 2.10 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 of execution. This is illustrated 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).

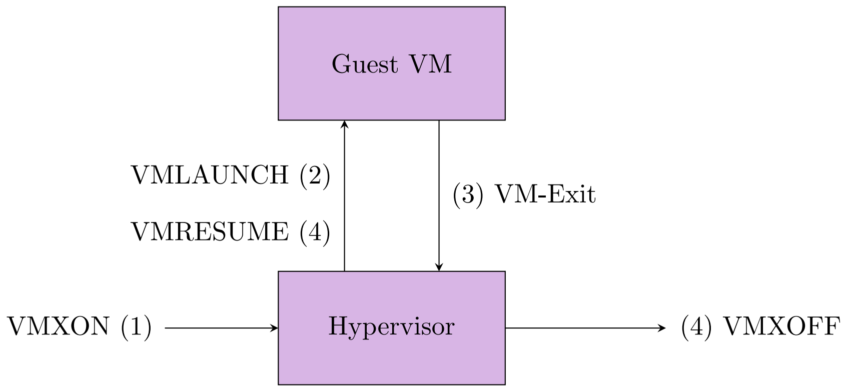

Figure 2.10: Successful Hypervisor Life Cycle Under Intel VMX

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 OS exists in kernel space (ring 0). By design, user space processes can not 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 OS utilizes. Each system call has a vector that uniquely maps itself to system calls. 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 handler to process a request. As previously mentioned, there are two pairs of system call instructions defined by x86_64 CPUs: SYSCALL/SYSRET and SYSEN-

TER/SYSEXIT.

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 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 type 1 hypervisor. KVM was merged into the official mainline Linux kernel in version 2.6.20, which was released on February 5, 2007. Since its inception into the 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.

KVM is structured as a character device file named "/dev/kvm", which can be used as an API to interact or maniplate KVM VMs. In order to access this API, one must make use of the ioctl system call. The ioctl system call takes a file descriptor and a request as arguments. The KVM API provides users with dozens of ioctl-based requests that can be used to interact and/or maniplate a KVM VM. Some of the relevant requests include but are not limited to KVM_CREATE_VM, which creates a new guest VM, KVM_RUN, which is a wrapper to the VMX defined instruction "VMLAUNCH", KVM_GET_MSR, which returns a value for a specific guest MSR, and KVM_SET_MSR, which can be used to set a value of a specific guest MSR. Notable user space VM management tools like libvirt and virt-manager make use of the KVM API to manage KVM VMs for users.

The KVM kernel module can not, by itself, create a VM. To do so, it must use QEMU, a VMX root mode binary called qemu-system-x86_64. As QEMU is a user space process, it also utilizes the /dev/kvm character device file as an API to interact and/or maniplate a KVM VM. For example, QEMU uses KVM_CREATE_VM ioctl request to create a VM. There is one QEMU process created for each guest. Therefore, 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 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 Linux user space processes. 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 can not handle an instruction generated by a KVM guest. QEMU is able to achieve hardware emulation by using Tiny Code Generator (TCG), which is a Just-In-Time (JIT) compiler that transforms an instruction written for a given processor to another one. Therefore, KVM lets 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 defined by the host CPU. If an instruction executed by a guest results in a VM-exit, and is not supported defined by the physical CPU, then QEMU will use the TCG to translate and execute the instruction on behalf of the guest if and only if TCG is enabled. If TCG is not enabled, then QEMU can not emulate any guest instructions. To aid in understanding the life cycle of a KVM VM, we present and explain an example (Figure 2.11) to illustrate how QEMU and KVM would handle an arbitrary instruction X that results in a VM-exit. In Figure 2.11, we assume that TCG is enabled.

First, a character device file named /dev/kvm is created by KVM (1). As previously mentioned, (1) allows QEMU to utilize the character device file to make requests to the KVM hypervisor. In our example, a user requested to begin execution of a specific guest. Thus, QEMU makes an ioctl system call with argument KVM_RUN to instruct KVM to start the guest (2). Internally, KVM will esecute the VMXON instruction. Afterwards, KVM will begin executing the guest by calling VMLAUNCH (3). The guest will now run until it requires the hypervisor to execute an instruction on its behalf. In our example, the guest attempts to execute an arbitrary instruction X, however, it is unable to (4). 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 instruction

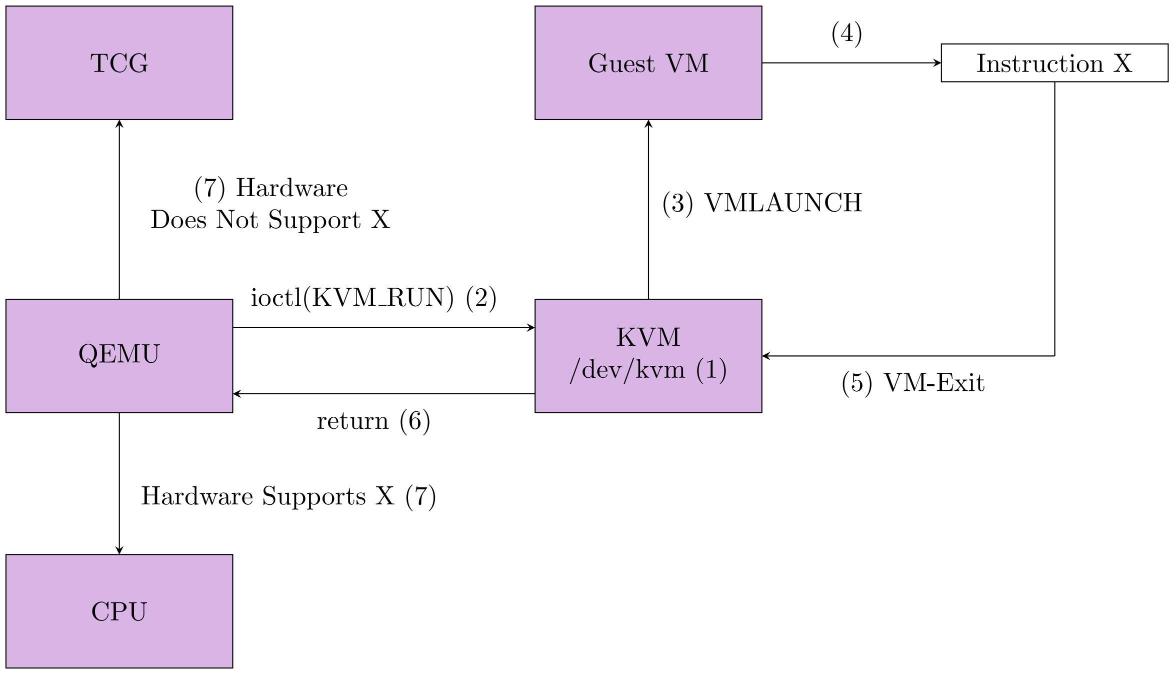

Figure 2.11: Decision on Whether QEMU use TCG or CPU for Executing an Arbitrary Instruction X.

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). After instruction X is successfully executed by either the CPU or TCG, QEMU will make another ioctl system call to request KVM to continue guest execution. In other words, a VM-entry will be made to switch contexts from VMX root to VMX non-root.

We must now consider the case where TCG is disabled either implictly (due to QEMU default settings) or explictly (by the user). In any case, if TCG is disabled, then QEMU will try to direct the guest instruction to the machine's CPU for execution. However, what if the instruction is not defined by the machine's CPU? In the case where TCG is disabled and a machine's CPU can not execute a guest instruction, then KVM will attempt to utilize the Linux kernel to emulate the instruction. The Linux kernel implements a small amount of functions that is able to emulate a non exhaustive amount of Intel x86 instructions. For example, the following code is the Linux kernel function that emulates the SYSCALL function.

```
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 to call em_syscall when both the CPU and TCG can not execute the SYSCALL instruction? Firstly, 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 guest 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 to emulate the SYSCALL/SYSRET pair of instructions. Afterward successful emulation of the SYSCALL instruction, a VM-entry will be made with a VMRESUME instruction. In figure 2.12, we assume that TCG is disabled, and the SYSCALL instruction is undefined becuase the SCE bit of the IA32_EFER MSR is unset.

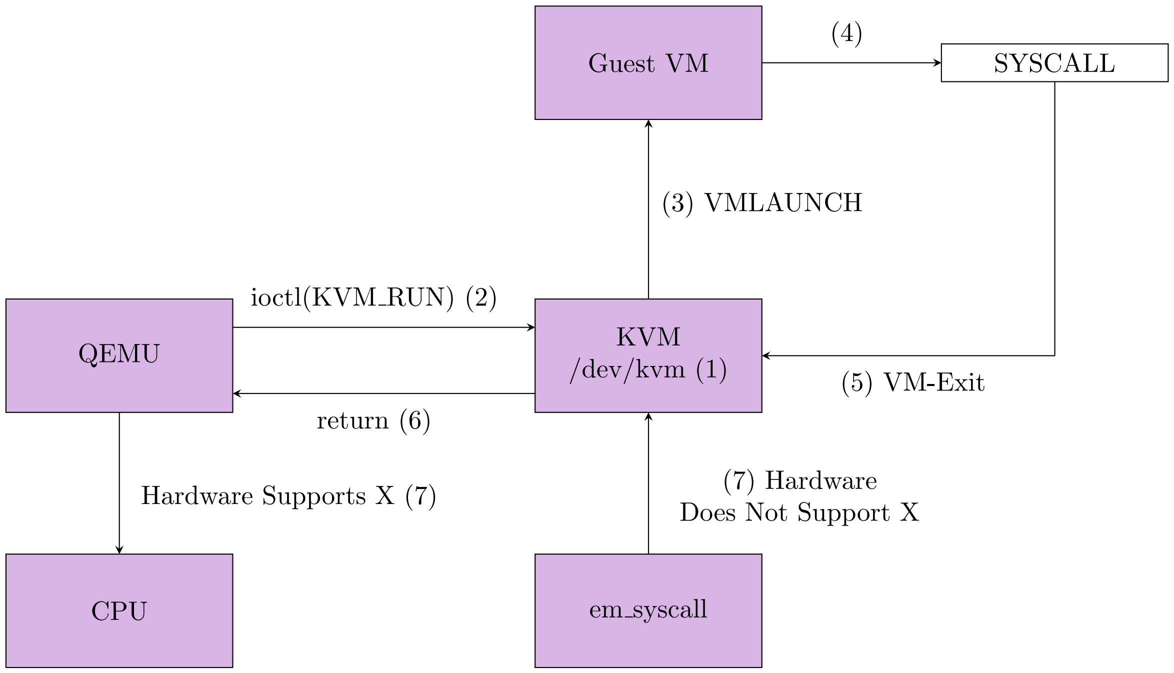

Figure 2.12: Partial KVM Life Cycle if TCG is Disabled

The worse case of any guest is if all three scenarios are true in the event of 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 Linux kernel is not able to emulate the instruction.

If any guest comes across this scenario, then it will halt forever because the instruction will never be executed. In this case, the VM must be rebooted, or a previous save state must be loaded.

2.6 eBPF

2.6.1 Overview

eBPF is a native Linux 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 software to provide an overview of the running system. 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 are non-native Linux programs running 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 was released in Linux kernel version 4.4.

2.6.2 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 a front end API for eBPF that allows users to write eBPF programs in C while using high level languages like Python, Go, and C++. When these programs are run, BCC will generate eBPF bytecode from C, and submit it to the kernel. Before the bytecode is loaded into the kernel, the eBPF program must pass a certain set of requirements. This involves executing the eBPF program to a verifier that performs 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 [9]. Other checks the verifier makes 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 [22], and verifies that no out of bounds memory accesses are possible [9]. If all checks pass, the eBPF program is sent to a JIT compiler that translates the eBPF bytecode into machine specific instructions in order to optimize execution speed of the program [22]. Afterwards, the eBPF program is loaded into the kernel, and begins to listen for kernel events that the eBPF program specified will 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). In all cases, from a high level, kernel events work by hooking functions that need to be observed. When desired functions are hooked, a function call is added to the code. This is illustrated in Listing 2.4 and Listing 2.5. When the hooked function is called, the eBPF program will capture and transfer its data to user space, allowing us to simply observe the data or manipulate it. The transfering and storage of event data from kernel space to user space is achieved by a generic data structure called maps. Due to its characteristic of being generic, eBPF 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 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 explain uprobes or kprobes further, because they are not utilized in our VMI system.

```
void func(int num){
    sum +=num
}
```

Listing 2.4: Function func that is not hooked

```
void func(int num){
    func_triggered() // function that the hook implemented.
    sum +=num
}
```

Listing 2.5: Function func that is hooked

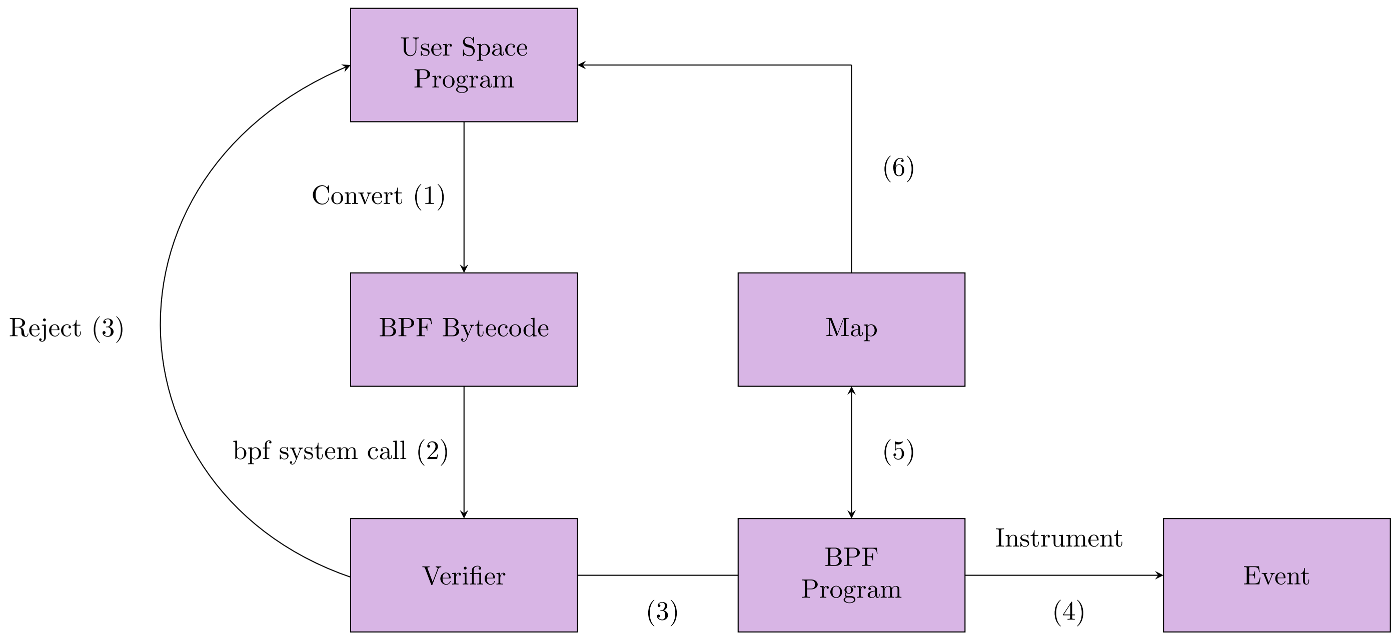

Figure 2.13: Illustration of eBPF Life Cycle

2.7 The Linux Kernel Tracepoint API

2.7.1 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 like eBPF for kernel debugging and performance problem diagnosis. 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 possible to extend the tracepoint API to hook kernel functions that are of interest to you.

2.7.2 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	

2.7.3 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.

2.7.4 Tracepoint Format File

Each event has a format file that provides a user with documentation about the data that is tracable for a particular event. For example, the format file for KVM exits can be found in /sys/kernel/debug/tracing/events/kvm_exit/format, and is also shown in Listing 2.6 for your convenience. The arguments shown in the format file are populated by mainline kernel space data. For example, in the case of the kvm_exit event, the variables presented in the format file are populated by data from the kernel space KVM function __vmx_handle_exit. When eBPF traces the kvm_exit event, the information that these arguments contain is what is stored and sent to user space via the eBPF map generic data structure.

```
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 2.6: Format File for the kvm_exit Linux Kernel Tracepoint Event — Linux kernel V5.18.8

2.7.5 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, for 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 those 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 shown in Listing 2.6.

TP_fast_assign

TP_fast_assign consist of the kernel space variables that instantiate the fields found in the format file.

TP_printk

TP_printk is responsible for using the field values in the format file to display a relevant tracing message to programs like eBPF.

2.8 Intrusion Detecion Prevention System

2.8.1 Overview

With respect to our thesis, intrusion detection is the process of monitoring events that occur in a computer system, and analyzing them for anomalies. An anomaly is a system violation that is pre-defined by the intrusion detection system. 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 anomalous activity. An IDPS is a software that automates the intrusion detection process, and also responds to anomalous activity by attempting to stop it from continuing. In this section, we will only write about IDPS. IDPS software can use one of two notable methodologies to detect anoamlies: signature-based, and anomaly-based.

2.8.2 Signature-Based Detection

A signature is a pattern that corresponds to a known attack in the field of computer security. 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.

2.8.3 Anomaly-Based Detection

Anomaly-based detection is the process of identifying significant changes to a computer systems state by comparing pre-defined definitions of what activity is considered normal to the current state of a computer system. An IDPS using anomaly-based detection has profiles that represent the normal behavior of a specific computer event. 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 anomalous, and respond to it. 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 system calls that deviate from the normal profile. Despite the fact that it is new malware, anomaly-based detection can detect and respond to it as long as it deviates from a normal profile. A disadvantage of anomaly-based detection is that building profiles that distinguish normal behavior from anomalous behavior is very challenging because computing activity is complex, always changing. As a result, it is difficult to predict what constitutes normal behaviour with high accuracy. For example, if a particular program K performs a special normal activity M once a month that consists of executing a large number of system calls. The activity M may not have been observed during the training period of a normal profile for program K. Therefore, when the normal activity M occurs, it is likely that the IDPS will consider it anomalous because it significantly deviates from the existing normal profile. For that reason, the IDPS will incorrectly respond to program K, causing normal behavior to fall victim to the IDPS.

2.9 Virtual Machine Introspection

2.9.1 Overview

The term "Virtual machine introspection" (VMI) was introduced by Garfinkel and Rosenblum in 2003 in their paper on IDS [10]. They defined VMI as a method for monitoring and analyzing the state of a VM from either the hypervisor or the guest. Today, VMI systems are almost always implemented on the hypervisor due the reasons mentioned in Section 1.4.2 [4]. VMI systems fall into one of two categories: those that only monitor behavior, and those that interfere with the guest based on certain conditions [23]. For example, a previously implemented VMI system named Nitro monitors VMs to gather system call and process information. When it detects system calls and processes, it merely reports it to the user, and does not interfere with the guest. In contrast, our VMI system is intended to gather system call and process information, as well as intefere with the guest upon the detection of an anomalous system call.

2.9.2 The Semantic Gap Problem

The primary advantage of in-VM systems is their direct access to OS level abstractions. However, when using a hypervisor-based VMI, access to these OS level abstractions are absent from the hypervisor. Although hypervisors have a full view of the entire state of the VMs they monitor, it consists of data that is not easily readable by human due to their lack of context [1]. Therefore, there is a semantic gap between what we can observe from the hypervisor and what we want [1]. We must partially bridge the semantic gap to provide VM monitoring services from the hypervisor.

Related Work

In this section, we will write about Nitro and pH, which are two research projects that is related to my thesis.

3.1 Nitro

3.1.1 Overview

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 two pairs of system call instructions (SYSCALL/SYSRET, SYSENTER/SYSEXIT) 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.

3.1.2 Properties of Nitro

Nitro's Mechanism for Tracing System Calls From The Hypervisor

As mentioned in our first research question, KVM is a type 1 hypervisor (see Section 1.3). Thus, guest instructions that are defined by the CPU (like system call instructions) 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 in order to trace VM system call events, instructions must be explictly trapped and emulated at the hypervisor level instead of being sent directly to the CPU for execution. In some cases, virtualization extensions like VMX, supported trapping specific instructions 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 VMX. Because Nitro is intended to work on both 32-bit and 64-bit systems, it is required to trap and emulate all two pairs of system call instructions defined by the x86 archetecture. As both pairs of 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 IA32_EFERs MSR (See section 2.2.11). Thus, the SYSCALL and SYSRET instructions can effectively be defined and undefined by setting and unsetting the System Call Extension (SCE) bit in the IA32_EFER MSR, respectively. A guest 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 register does not indicate that the #UD exception was caused by either SYSCALL or SYSRET instruction, then Nitro injects an invalid opcode exception into the guest OS and performs a VM-entry. However, if the %rip register indicates that a SYSCALL or SYSRET instruction was executed, then Nitro collects the desired information, emulates the instruction, and returns control back to the guest with a VM-entry.

Process Identification

Nitro's goal was to not only trace system call instructions, but to also trace and determine which process produced a system call. How does Nitro do this? They collect and store the value of the CR3 register in their database. As explained in section 2.2.3, the CR3 register can identify a process due to the fact that it is unique to each process.

How Nitro Empowers Anomaly Detection

Nitro's implementation allows for tracing guest system calls From the host. However, Nitro does not monitor for anomalous systems calls, nor does it respond to them. Instead, Nitro expects external applications to utilize Nitro's system call and process tracing capabilities to monitor and respond to anomalies. Different external applications for system call and process monitoring want a varying amount of information. In some cases, an application may only want a simple sequence of system calls, while other application may require more detailed information complex sequences of system calls, and process information. As Nitro cannot foresee every need of external applications that conduct system call monitoring and response, it does not deliver a fixed set of data per system call to the user. Instead, it allows the user to define flexible rules to control the data collection during system call and process tracing. Nitro will then extract the system call and process information based on the user's specification. With these capabilities, Nitro can be used effectively by a variety of third party IDS and IPS.

3.2 pH

3.2.1 Overview

In Dr. Somayaji's dissertation [30], he introduced an extension to Linux kernel version 2.2 called pH (process Homeostasis), which is a program that traces system calls, monitors them for unusual behavior, and responds to it by slowing down that behavior [30]. He mentions lookahead pairs or full sequences as two possible system call anomaly detection methods [15]. pH employs lookahead pairs for detecting anomalies within a short sequence of system call of a process [30]. Recall that a sequence is an ordered list of events in which repetition is allowed. When pH detects that a process is behaving unusually, it responds by slowing down that process's system calls. The goal of slowing down system calls of anomalous process's is to stop attacks before they can do damage, and to give users time to decide whether further action is needed [30]. What follows is how pH classifies normal profiles, and how it responds to sequences of system calls that deviate from the pre-defined normal profiles.

3.2.2 Classifying Normal Profiles

To determine whether a process is behaving abnormally, pH maintains a profile with two datasets (for every process): a training dataset and a testing dataset. A training dataset is the initial set of data that is used to train pH, so that it can make predictions on whether a sequence of system calls is anomalous or not. The testing dataset is initially empty, and the profile for a process is considered abnormal because it does not accurately represent the program's normal behaviour [30]. Before pH can begin detecting anomalies, it must have a valid testing dataset. How does pH acquire a valid testing dataset? pH continuously adds the lookahead pairs associated with a process's current system call sequence to its training dataset. When no new pairs are added to a training dataset for an extended period of time, or at the request of a user, the training dataset is copied over to the testing dataset [30]. Therefore, a testing dataset can be defined as both a subset of the training dataset, and the default dataset of pH that is utilized to evaluate whether a process's system call sequences are anomalous or not.

3.2.3 Responding to Anomalous Process's

As mentioned in the overview section, pH responds to anomalous system calls by delaying its process with the assumption that the process will produce more anomalous system calls if it is not delayed. With IPS, there is always the risk of false positives, in which normal processes are flagged as abnormal. To avoid false positives from affecting normal process's too harshly, pH scales its response in proportion to the number of recent anomalies detected for a given process [9]. For example, if pH triggers several dozen anomalies for a given process, it will be slowed down to the point of effectively stopping it. In contrast, if pH triggers only once or twice for a given process, it may only be delayed by a few seconds. For this reason, a false positive is less likely to affect a normal process to the point of termination. However, this is based on the assumption that the number of false positives that occur are low. This assumption generally holds due to pH's use of lookahead pairs and not full sequences of system calls. This is because Somayaji claims that using lookahead pairs for monitoring and detecting anomalous sequences of system calls results in fewer false positives than the full sequences method [9].

3.3 LibVMI

LibVMI is a powerful and flexible library that allows researchers and developers to access and analyze the memory and state of a virtual machine from the host operating system. It provides a high-level programming interface that simplifies the task of VMI and facilitates the development of advanced security and forensic tools. With LibVMI, it is possible to perform a wide range of tasks, including memory forensics, malware analysis, vulnerability detection, and intrusion detection. The library supports a variety of virtualization platforms, including Xen, KVM, and VMware, and provides a range of features and capabilities, including process and thread management, file system analysis, and network traffic interception. One of the key benefits of LibVMI is its flexibility and extensibility. The library is open-source and can be customized and extended to meet the needs of specific use cases or research projects. It also has a large and active community of developers and users, who contribute to its ongoing development and improvement.

Unlike Frail, LibVMI uses debugging symbols to help map the virtual machine's memory to the corresponding code and data structures in the guest operating system. Debugging symbols are metadata that are generated by a compiler or linker when an executable or library is built. They contain information about the names and types of variables, functions, and other code and data structures in the binary, as well as information about their memory addresses and layout. To use debugging symbols, LibVMI first obtains a copy of the guest operating system's kernel image and any relevant shared libraries. It then uses the debugging symbols associated with these binaries to map the virtual machine's memory to the corresponding code and data structures. By using debugging symbols, LibVMI can provide more detailed information about the guest operating system's memory and state, including the names and values of variables and the parameters and return values of function calls. This information can be useful for debugging, profiling, or reverse-engineering purposes. However, it is important to note that not all binaries may have debugging symbols available, and even if they do, they may not be accessible or usable by LibVMI. Additionally, using debugging symbols can introduce additional overhead and complexity, so they should be used judiciously and only when necessary.

Contributions & Improvements On Related Work

To summarize, our contributions are as follows:

- Nitro's implementation only allows tracing of system calls of 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 VM was created.
- Nitro uses Rekall (a memory forensics framework) to retrieve process information from VMs. Rekall is discontineud and not no longer maintained. Frail uses our own implementation to retrieve guest process information by reading the %rdi register everytime an exec family of system calls is executed on a guest. This way, we don't have to rely on third party software to retrieve process information.
- We extend the Linux kernel tracepoint API by defining two new kernel events: (1) KVM guest system calls and (2) guest process's. The API extension allows eBPF programs to instrument these two events VMX root kernel space to VMX root user space.
- Nitro is not capable of monitoring and responding to anomalous guest system calls. With our prototype, we provide the ability to monitor anomalous sequences of system calls using a modified version of pH. We respond to anomalous sequences of system calls by either terminating the process responsible for the anomalous system call, or shutting down the relevant VM entirely. Essentially, we are including an IPS into our VMI systtem. Our intent is to monitor and respond to anomalous sequences of system calls in real time without hindering the usability of the guest.

Designing Frail

In this section, we introduce the design of our KVM and Intel VMX exclusive hypervisor-based VMI system called *Frail*. More specifically, we discuss the design of the four notable components that make up our VMI. Firstly, we explain how our VMI intends to trace every SYSCALL instruction from any running KVM VM. Secondly, we explain how our VMI system intends to trace guest process's that make system calls. Thirdly, we explain how we integrate Somayaji's pH implementation with our VMI, in order to monitor sequences of system calls for anomalies. Lastly, we explain how we intend to respond to anomalous sequences of system calls that are caught by our IPS.

5.1 Tracing KVM VM System Calls

With the advent of VMX, a majority of KVM guest instructions were able to 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, the SYSCALL/SYSRET pair of system call instructions are defined by modern Intel x86 CPUs (see 2.2.10). Thus, they do not require a VM-exit, and execute 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 non-root to VMX root.

5.1.1 Trapping SYSCALL/SYSRET from VMX Non-Root to VMX Root

Our design for tracing the KVM guest SYSCALL instructions is based on the method of trapping and emulating the instructions such that it results in a VM-exit to VMX root. In this subsection, we describe how we trap SYSCALL/SYSRET instructions. First, we unset the SCE bit of the IA32_EFER MSR using the WRMSR instruction prior to every VM-entry. We require unsetting the SCE bit prior to every VM-entry to ensure that the SCE bit will be unset before a subsequent SYSCALL/SYSRET instruction takes place. Secondly, we also want to ensure that the SCE bit is unset for the first state of the VM (e.g. when the VM is first powered on, or a save state is loaded into memory). Recall (from section 2.2.10) that whenever the SCE bit of the IA32_EFER MSR is unset, every SYSCALL/SYSRET instruction executed in the guest will result in a #UD exception. Again, recall (from section 2.2.2) that a #UD exception is a fault that traps execution from user space to root mode. In the case of a VM, a #UD exception is a fault that traps execution from VMX non-root mode to VMX root mode, so that the fault can be handled appropriately by the KVM hypervisor. Thus, by unsetting the SCE bit of the IA32_EFER MSR, we are able to trap every SYSCALL/SYRET instruction that is executed in a KVM VM to VMX root. The method we described only traps SYSCALL and SYSRET instructions.

5.1.2 Emulating SYSCALL & SYSRET Instructions

Once a VMs SYSCALL/SYSRET pair of instructions trap to VMX root, we are given two choices to handle the trap. We can utilize QEMU's TCG capabiltiies to emulate the instructions and VM-entry back into the KVM VM. In contrast, we can utilize KVM's emulation capabilities (see Section 2.5) to emulate the instruction, 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 own function that emualtes this instruction (see Figure 5.1). 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 SYSRET emulation function. The emulation function for the SYSCALL instruction is already implemented and handles the case for #UD exception caused by SYSCALL. Thus, no further changes need be made to KVM.

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

    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);
    }

    if(ctxt->ops->cpl(ctxt) != 0){
        return emulate_gp(ctxt,0);
    }

    rcx = reg_read(ctxt, VCPU_REGS_RCX);
    if(( (rcx & 0xFFFF800000000000) != 0xFFFF800000000000) && ( (rcx & 0x00007FFFFFFFFFFF) !=
        rcx)){
        return emulate_gp(ctxt,0);
    }
    ops->get_msr(ctxt, MSR_EFER, &efer);
    ops->set_msr(ctxt, MSR_EFER, (efer & ~1) | (0x00000001 & 1));
    ops->get_msr(ctxt, MSR_EFER, &efer);
    setup_syscalls_segments(&cs, &ss);
    if (!(efer & EFER_SCE)){
        return emulate_ud(ctxt);
    }

    ops->get_msr(ctxt, MSR_STAR, &msr_data);
    msr_data >>= 48;
    if (ctxt->mode == X86EMUL_MODE_PROT64){
      cs_sel = (u16)((msr_data + 0x10) | 0x3);
      cs.l = 1;
      cs.d = 0;
    }
    else{
      cs_sel = (u16)(msr_data | 0x3);
      cs.l = 0;
      cs.d = 1;
    }
    cs.dpl = 0x3;

    ss_sel = (u16)((msr_data + 0x8) | 0x3);
    ss.dpl = 0x3;

    ops->set_segment(ctxt, cs_sel, &cs, 0, VCPU_SREG_CS);
    ops->set_segment(ctxt, ss_sel, &ss, 0, VCPU_SREG_SS);

    ctxt->eflags = (reg_read(ctxt, VCPU_REGS_R11) & 0x3c7fd7);
    ctxt->_eip = reg_read(ctxt, VCPU_REGS_RCX);

    ops->set_msr(ctxt, MSR_EFER, (efer & ~1) | (0x00000000 & 1));
    return X86EMUL_CONTINUE;
}
```

Listing 5.1: Emulation of SYSRET instruction

5.1.3 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 guest system call of a particular VM, we unset the SCE bit of the IA32_EFER MSR for every VCPU that exists on the VM using the WRMSR instruction. We do this step for every KVM VM that exists on a system.

5.1.4 Verifiying a SYSCALL/SYSRET Caused a VM-exit

How do we know that a VM-exit was caused by a SYSCALL/SYSRET instruction, and not something else? In our design, two checks are made to verify this. Recall that every #UD exception causes a VM-exit with code 0. Therefore, we filter out VM-exits to include only 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 SYSCALL/SYSRET instruction. It can be caused by any undefined instruction to a CPU. Therefore, our second check consists of checking the %rip register. Recall that the %rip register 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). With these two checks, we can guarantee that the instructions that we trace is only a SYSCALL instruction. This approach allows a VMI system to introspect guest system call events in the ideal way: without hindering the usability of the guest and host (research question 3).

Algorithm 1 Algorithm to Verify Instruction is a SYSCALL

```
1: if $VMEXIT = 0$ then
2:     $Disable(SMAP)$
3:     $INS = Read(\%RIP)$
4:     if $(((INS >> (8*0))\&0xff) == 0x0f)$ && $(((INS >> (8*1))\&0xff) == 0x05)$ then
5:         $RAX$ = Read(%RAX)
6:         $Enable(SMAP)$
7:         return RAX
8:     end if
9:     $Enable(SMAP)$
10: end if
```

5.1.5 Extending the 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 0 of VMX root to ring 3 of VMX root. For that reason, we extend the Linux kernel tracepoint API by creating a new event. The event will be triggered every time the KVM system call emulation function em_syscall is called. We do not require data from the SYSRET instruction, thus, we do not create a tracepoint for it. As eBPF programs can utilize the Linux kernel tracepoint API, we can create a simple Python 3 eBPF program (using BCC library) to trace these system calls from userspace. Figure 5.1 illustrates this interaction, and Listing 5.2 illustrates the implementation of the KVM SYSCALL tracepoint.

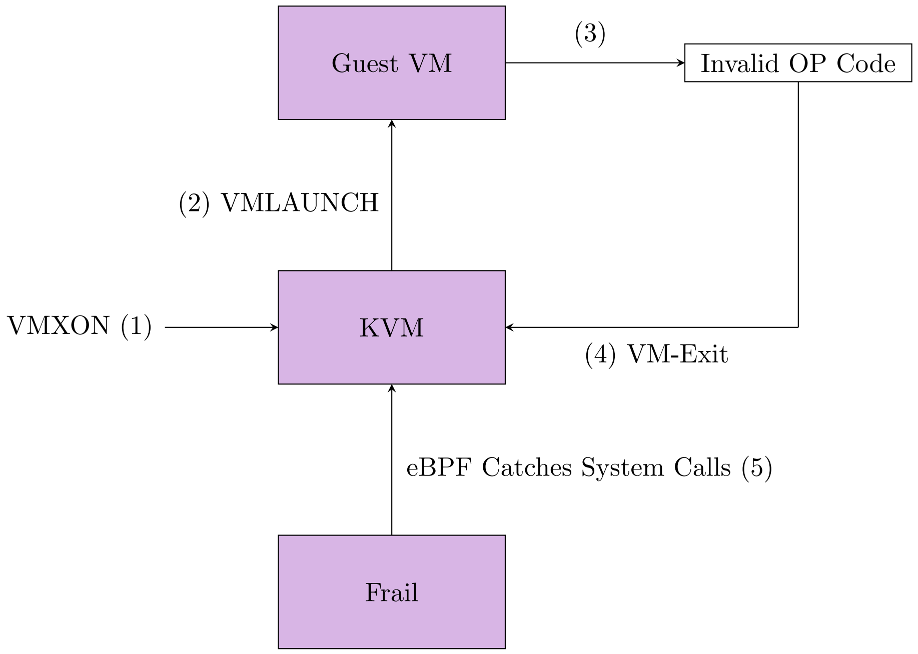

Figure 5.1: Illustration of Tracing KVM VM System Call

```
TRACE_EVENT(kvm_syscall,
    TP_PROTO(int pid, unsigned long cr3, unsigned long vector, int vcpu_number, char*
        process),
    TP_ARGS(pid, cr3, vector, vcpu_number, process),

    TP_STRUCT__entry(
        __field(int, pid)
        __field(unsigned long, cr3)
        __field(unsigned long, vector)
        __field(int, vcpu_number)
        __field(char*, process)
    ),

    TP_fast_assign(
        __entry->pid = pid;
        __entry->cr3 = cr3;
        __entry->vector = vector;
        __entry->vcpu_number = vcpu_number;
        __entry->process = process;
    ),

    TP_printk("pid=%d cr3=%lu vector=%lu vcpu_number=%d process=%s", __entry->pid,
         __entry->cr3, __entry->vector, __entry->vcpu_number, __entry->process)
);
```

Listing 5.2: Implementation of KVM SYSCALL Linux Kernel Custom Tracepoint

5.2 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 of a VM.

When a Linux process is executed, the CR3 register is loaded with the physical address of that process's page global directory (PGD). Because 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 systems because it means we do not have to constantly scan memory to keep track of which process is running. Instead, we can create a key, value pair data structure in user space of VMX non-root in which our keys are given by the CR3 register, and the values are given by system calls caused by the process corresponding to the CR3. Due to the guaranteed uniqueness of the value given by the CR3 register, multiple keys will not end up with the same hashcode. Thus, a collision will never occur.

The uniquenesss of CR3s help with storing process's and their corresponding system calls. However, simply tracking changes of the CR3 register does not give us enough insight into guest processes due to the semantic gap between the VM and hypervisor. More specifically, the CR3 register simply provides us with a unique address for every process; it does not give us the name of the process. 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). To get the process name, we utilize our system call tracer to track execve system calls. Why do we do this? Because to create a new proces, an execve system call is required. Also, the first argument of the execve system call requires the filename of the process. If we want to retrieve the filename that was passed as argument to exec, then we must read the %rdi register from the hypervisor. Recall (from section 2.2.6) that the %rdi register stores the virtual address of the 1st argument given to the execve system call 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 VM user space data from the hypervisor. KVM implicitly 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. As mentioned in section 2.2.11, copy_from_user temporarily disables SMAP. For this reason, we are able to successfully retrieve guest VM process information directly from the KVM hypervisor.

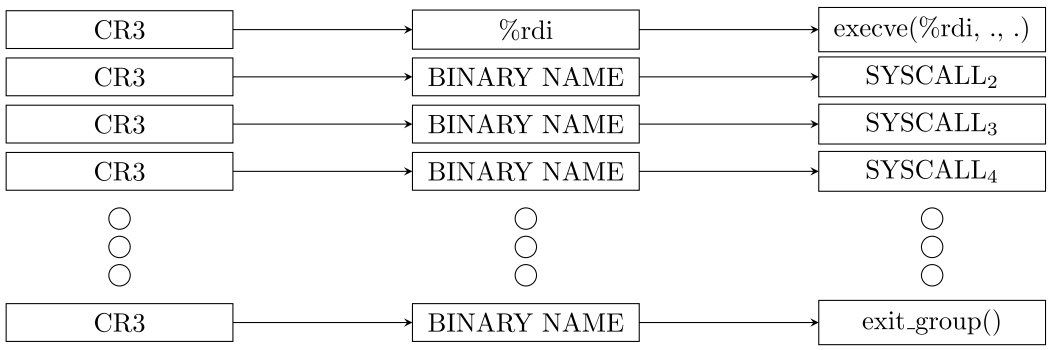

Figure 5.2: Relationship between CR3, %rdi, and SYSCALL
Note: BINARY NAME comes from %rdi

After implementing the logic to access 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 CR3 register 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 create a simple eBPF program to trace these two events.

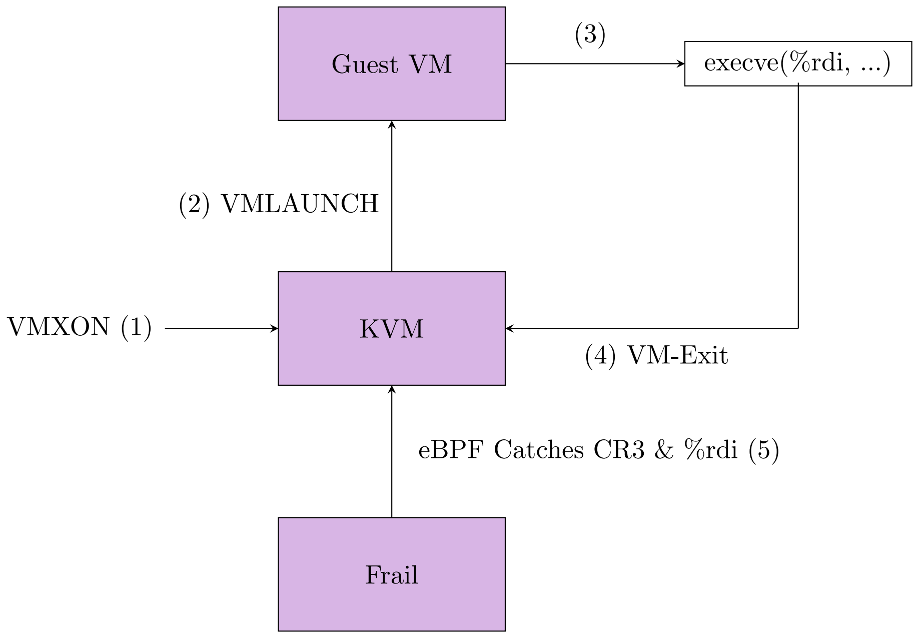

Figure 5.3: Illustration of Tracing KVM Guest Processes

As we are using KVM, Linux runs on the host and acts as the hypervisor, and hence it is the Linux scheduler that schedules the virtual CPUs of the KVM VMs onto the host's physical CPUs. In other word, the KVM VM VCPUs are processes running in VMX root. Therefore, the Linux OS scheduler is used to determine which process to run next. Therefore, it is upto the the currently running OS scheduler to determine which VCPU a system call is executed with. For example, let us define a process X that executed N (where $N > 1$) system calls, and a VM that has M (where $M > 1$) VCPUs. For each system call k, it is not certain that k will be executed on the same VCPU. Therefore, we must make sure that we are checking system calls for every VCPU we come across.

5.3 Generating Sequences of System Calls

UNIX processes are created via the fork system call, a binary is executed on the process using an execve system call, and a process is terminated using the exit_group system call. We use these three facts to create two datasets for every process: a training dataset and a testing dataset. The algorithm used to build the training dataset is trivial. We begin by tracing system calls generated by a particular process, and build up a database of all unique sequences of a given length. When no new sequences are added to a training dataset for an extended period of time t, or at the request of a user, the training dataset is copied over to the testing dataset. Therefore, each process will have a different training dataset.

For example, assume that the binary /usr/bin/ls generates the following system calls:

$$\{\text{open, read, mmap, mmap, open, read, mmap} \mid k = 3,\ t = 3 \text{ minutes}\}$$

Thus, we get the following sequence of system calls in our training dataset:

open, read, mmap
read, mmap, mmap
mmap, mmap, open
mmap, open, read

If we assume that no new sequences of systems calls of size k are executed by any processes running /usr/bin/ls for t minutes, then the testing dataset will equal to the training dataset. The training dataset will be used as the normal profile Otherwise, a new sequence of size k is added to the end of the training dataset, and the time t is reset to 0.

5.4 Storing Sequences of System Calls

In order to allow profile data to persist across machine reboots, Frail writes profile data to disk everytime a new sequence is added to a training or testing dataset of a process, or when the program is terminated. Profiles are read from disk when Frail first loads. In the original pH, profile data was saved and loaded in kernelspace [30] which meant that it required kernelspace file I/O, which is often regarded as an unsafe practice. Inspired by ebpH, Frail solves this problem by moving all file I/O operations into userspace. Specifically, when writing or reading a sequence from/to disk, Frail will write the data to a file defined by ({testing, training}_profile/<BINARY NAME>).

5.5 Measuring Anomalous Behavior

We monitor future process system call behavior, and compare it to their corresponding training dataset for significant deviations. We do this by looking at all future sequences of length k in the new trace, and determine if they are represented in the normal dataset. Sequences that do not occur in the training dataset are considered to be mismatches. By recording the number of mismatches, we can determine the strength of an anomalous signal [14]. Thus the number of mismatches occurring in a new trace is the simplest determinant of anomalous behavior. Ideally, we would like these numbers to be zero for new examples of normal behavior, and for them to jump significantly when abnormalities occur.

We make a clear distinction here between normal and legal behavior. Normal behavior of a program is considered to be a program flow of execution that is desirable or expected for a user that is not malicious. Legal behavior is program flow of execution that is possible, but not desirable or expected by a user that is not malicious. In the ideal case, we want the training dataset to contain all possible sequences of normal behavior, but we do not want it to contain every single possible path of legal behavior, because our design assumes that normal behavior forms a proper subset of the possible legal execution paths through a program, and unusual behavior that deviates from those normal paths signifies an intrusion or some other undesirable condition. We want to be able to detect not only intrusions, but also unusual conditions that are indicative of system problems. For example, when a process runs out of disk space, it may execute some error code that results in an unusual execution sequence (path through the program). Clearly such a program flow is legal (because it is a defined path by the program), but certainly it should not be regarded as normal because it is not desirable or expected by a non-malicious user.

If the training dataset does contain all variations of normal behavior, then when we encounter a sequence that is not present in the normal dataset, we can regard it as anomalous, i.e. we can consider a single mismatch to be significant. In reality, it is likely to be impossible to collect all normal variations in behavior, so we must face the possibility that our normal database will provide incomplete coverage of normal behavior, causing a false negative. One solution to this is to count the number of mismatches occurring in a trace, and only regard as anomalous those traces that produce more than a certain number of mismatches. This is problematic however, because the count is dependent on the length of the trace, which might be indefinite for processes that run continuously.

5.6 Responding to Anomalous Behavior

When our sequences of system calls algorithm detects a malicious process, we either terminate the process, or the associated 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.

5.6.1 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. In fact, Frail automates anomaly pevention by using the kill system to terminate a KVM VM. If a user wanted to manually terminate a KVM VM, one could also use the kill system call, or use the virtual machine manager software to terminate the virtual machine.

5.6.2 Terminating Malicious Process

Terminating a guest process from the hypervisor is not as trivial as terminating a VM. Like other non-trivial approaches needed for hypervisor-based VMI systems, this is one is also due to the semantic gap. When a malicious process makes a system call, we will replace the system call with the exit_group system call before VM-entry to the VM. We do this by writing 231 to the %rax register. More specifically, we replace the system call number that %rax points to with the vector 231, which corressponds to the exit_group system call in the Linux system call for x86-64 CPUs. Recall that the exit_group system call is used to terminate all threads of the calling process. Due to the semantic gap, the KVM hypervisor does not have access to the PID of the process that executed the system call. However, unlike the kill system call, the exit_group 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 Frail detected as anomalous. After terminating the process, the %cr3 value that coresponds to the process will be removed from our database of known KVM VM processes.

Testing Frail

6.1 Overview

One of the primary advantages of Frail is its ability to trace process system calls from the hypervisor, without needing to modify the VM that is being introspected. In order to justify this claim, it is necessary to ascertain Frail's ability to trace system calls for a variety of processes under a variety of workloads (artificial and otherwise). Therefore, in this chapter, I describe the tests that were conducted in order to determine Frail's ability to trace system calls. Section 6.1.1 outlines the systems and tools used for testing, and provides an overview of the collected datasets. The specifics of each test along with the results are provided in Section 6.2. Frail is also a detection system. Thus, we must also adequately show that Frail is able to detect anomalies whenever a process deviates from its normal profile dataset. Therefore, in section 6.3, we provide a simple example of of Frail's anomaly detection capabilities.

6.1.1 Methodology

Table 6.1 details more information about the system used for the collection of processs system calls.

Table 6.1: System used for the collection of process system calls

System	Description	Specifications	
`Ubuntu Linux`	Personal workstation	Kernel:	5.18.18
		CPU:	Intel(R) Core(TM) i7-8550U @ 1.80GHz
		GPU:	UHD Graphics 620
		RAM:	16GB DDR4
		Disk:	1TB SSD
`Ubuntu Linux`	KVM Virtual Machine	Kernel	5.18.8
		CPU	Intel(R) Core(TM) i7-8550U @ 1.80GHz
		#VCPUs:	4
		GPU	UHD Graphics 620
		RAM	4GB DDR4
		Disk	80GB

6.2 Testing Frail's Ability to Trace Process System Calls

6.2.1 Validating Sequences of System Calls

To test the correctness of Frail, we make a simple bash script that executes all the binaries inside of /usr/bin by giving it to strace as an argument. The bash script will also create sequences of system calls of length k for each binary, and place them in a file. Before executing the bash script inside a KVM VM, we execute Frail in VMX root. After the bash script ends, we wait for Frail to create system call sequences of length k of all the binaries that were executed by strace (in the bash script). Finally, we compare the sequences of system calls made by Frail with those made by the bash script, to see if they match. The importance of this test is that it verifies that Frail is able to trace every VM system call while it is running in VMX root. We assume that strace (when run in VMX non-root) is able to trace every system call of each binary given as argument. Therefore, we only need to test Frail. Listing 6.1 is the code that builds the sequences of system calls from strace output in KVM VM. We ran a total of 2399 binaries on our KVM VM based test bash script, and for each binary, Frail was successfully able to generate the same system call sequences as the bash script. We repeated this test 1000 times.

Algorithm 2 Comparing Sequence of System Calls (SOS) Generated by Frail with Bash Script SOS

```
1: repeat
2:     for each binary ∈ /bin/bash do
3:         if compare(SOS(binary), FRAIL_SOS) then return True
4:         return False
5:         end if
6:     end for
7: until binary = NULL
```

Table 6.2: Example of Binaries tested using Frail and Bash Script

(Full Listing Shown in Appendix)

Sequences by Frail	Sequences by Bash Script	Do they match?
/usr/bin/411toppm	/usr/bin/411toppm	YES
/usr/bin/7z	/usr/bin/7z	YES
/usr/bin/7za	/usr/bin/7za	YES
/usr/bin/7zr	/usr/bin/7zr	YES
/usr/bin/a2ping	/usr/bin/a2ping	YES
/usr/bin/a5toa4	/usr/bin/a5toa4	YES
/usr/bin/aa-enabled	/usr/bin/aa-enabled	YES
/usr/bin/aa-exec	/usr/bin/aa-exec	YES
/usr/bin/aa-features-abi	/usr/bin/aa-features-abi	YES
/usr/bin/aconnect	/usr/bin/aconnect	YES
.	.	.
.	.	.
.	.	.
.	.	.
/usr/bin/ls	/usr/bin/ls	YES
.	.	.
.	.	.
.	.	.

Continued on next page

Table 6.2 – Continued From Previous Page

Sequences by Frail	Sequences by Bash Script	Do they match?
/usr/bin/zstdmt	/usr/bin/zstdmt	YES

6.2.2 Validating System Calls

To test the correctness of Frail, we make a simple bash script that find all system calls of the binaries inside of /usr/bin by giving it to strace as an argument. We do this in the KVM VM that we are testing. We also add the '-n' option to the strace command to provide us with the system call number (instead of the name of the system call). For each binary that we use strace on, we will write the system call numbers to a corressponding file. Before executing the bash script inside the KVM VM that we are testing, we execute Frail in VMX root, so that it can wait for the system calls generated by strace from the bash script. After executing the bash script, we check to see if the system calls traced by Frail are the same as the ones found by strace. We ran a total of 2399 binaries on our KVM VM based test bash script, and for each binary, Frail was successfully able to generate the same system calls as the bash script. We repeated this test 1000 times.

6.2.3 Results: Testing Frail's Correctness

Table 6.3: Testing Traceability of /usr/bin/ls

System Calls Found by Frail	System Calls Found by strace	Do they match?
59	59	YES
12	12	YES
158	158	YES
9	9	YES
21	21	YES
257	257	YES
262	262	YES

Continued on next page

Table 6.3 – Continued From Previous Page

System Calls Found by Frail	**System Calls Found by strace**	**Do they match?**
9	9	YES
3	3	YES
257	257	YES
0	0	YES
262	262	YES
9	9	YES
10	10	YES
9	9	YES
9	9	YES
9	9	YES
9	9	YES
3	3	YES
257	257	YES
0	0	YES
17	17	YES
17	17	YES
17	17	YES
262	262	YES
17	17	YES
9	9	YES
9	9	YES
9	9	YES
9	9	YES
9	9	YES
3	3	YES
257	257	YES
0	0	YES
262	262	YES
9	9	YES
9	9	YES

Continued on next page

Table 6.3 – Continued From Previous Page

System Calls Found by Frail	System Calls Found by strace	Do they match?
9	9	YES
9	9	YES
3	3	YES
9	9	YES
158	158	YES
218	218	YES
273	273	YES
334	334	YES
10	10	YES
10	10	YES
10	10	YES
10	10	YES
10	10	YES
302	302	YES
11	11	YES
137	137	YES
137	137	YES
318	318	YES
12	12	YES
12	12	YES
257	257	YES
262	262	YES
0	0	YES
0	0	YES
3	3	YES
21	21	YES
257	257	YES
262	262	YES
9	9	YES
3	3	YES

Continued on next page

Table 6.3 – Continued From Previous Page

System Calls Found by Frail	System Calls Found by strace	Do they match?
16	16	YES
16	16	YES
257	257	YES
262	262	YES
217	217	YES
217	217	YES
3	3	YES
262	262	YES
1	1	YES
1	1	YES
1	1	YES
3	3	YES
3	3	YES
231	231	YES

6.3 Testing Anomaly Detection

Frail profiles are tracked in two phases, training mode and testing mode. Profile data is considered training data until the profile becomes normal (as described in Section 2.8.3). Once a profile is in testing mode, the sequences of system calls generated by its associated processes are compared with existing data. When mismatches occur, they are flagged as anomalies which are reported. The detection of an anomaly also prompts Frail to remove the profile's normal flag and return it to training mode. As an example, consider the simple program shown in Listing 3.5. This program's normal behavior is to simply return 0. However, when issued an extra argument (in practice, this could be a string to abuse a vulnerability), it executes an extra system call. This will cause a change in the sequences of system calls associated with the program's profile, and this will be flagged by Frail if the profile has been previously marked normal.

In order to test this, I artificially lowered Frail's normal time to three seconds instead of one week. Then, I run the above test program several times with no arguments to establish normal behavior. Once the profile has been marked as normal, I then run the same test program with an argument to produce the anomaly. Frail immediately detects the anomalous system calls and flags them. Once the anomaly is detected, the user also has the ability to terminate every future execution of the anomalous program or terminate the affected KVM VM. These anomalies are reported to userspace as shown in Listing 6.2.

```c
// anomaly_test.c
#include <unistd.h>
#include <sys/syscall.h>

int main(int argc, char** argv){
    if(argc > 1){
        syscall(158);
    }

    return 0;
}
```

Listing 6.1: A simple C program to demonstrate anomaly detection in Frail.

```
Warning: Found anomaly in anomaly_test
```

Listing 6.2: A simple C program to demonstrate anomaly detection in Frail.

6.4 VM-Exit Benchmark for Exception_NMI & WRMSR

In this section, we compare VM-exit benchmarks between the unmodified Linux kernel version 5.18.8 and our modified Linux kernel version 5.18.8. There are two key difference between these kernels in relation to vm-exits, The modified kernel performs a vm-exit for every SYSCALL and SYSRET instruction due to them being undefined. Therefore, both these instructions cause an ”exception or non-maskable interrupt” (NMI) VM-exit (reason 0). Also, after every vm-entry, a WRMSR instruction is forced to write a 0 on the SCE bit of the IA32_EFER MSR, causing a WRMSR vm-exit (reason 32). Thus, we only focus on these two vm-exits. Table 8.1 shows a comparison between the number of VM-exits caused by EXCEPTION_NMI (reason 0) and WRMSR (reason 32) with and without the modified kernel. The vm-exits were observed for a period of 24 hours. We can see that with the modified kernel (that Frail uses), the number of vm-exits caused by EXCEPTION_NMI increased by 3665590%, and the number of vm-exits caused by WRMSR increased by 30208300%.

Table 6.4: Number of EXCEPTION_NMI and MSR_WRITE VM-EXITS

VM-EXIT REASON	UNMODIFIED KERNEL	MODIFIED KERNEL	% Change
EXCEPTION_NMI (0)	12,600	461,876,928	+3,665,590%
MSR_WRITE (32)	24	7,250,016	+30,208,300%

vm-exits consume system resources, such as CPU cycles and memory bandwidth. By minimizing vm-exits, you reduce the overhead associated with these operations, allowing more resources to be used for other tasks. Thirdly, it improves scalability: When a large number of virtual machines are running on a system, the frequency of VM exits can become a bottleneck. By minimizing VM exits, you can improve the scalability of the system and support more virtual machines without sacrificing performance. Lastly, it improves security. In summary, minimizing VM exits is essential for achieving optimal system performance, scalability, and security in virtualized environments.

Shortcomings of Frail

7.1 Forcing VM-exits is Slow

VM-exits occur when a VM needs to request a service or resource from the hypervisor. This can include actions like accessing I/O devices, memory management, or VM management tasks like switching between different VM. Each vm-exit is an expensive operation that requires a significant amount of processing time, which can negatively impact system performance. As presented in the previous chapter, a vm-exit is required for each SYSCALL, SYSRET, and WRMSR instruction. As a result, we present a number of justifications for why forcing vm-exits to trace system calls and processes is a shortcoming of Frail. Firstly, vm-exits are slow because they require a context switch from the VM to the hypervisor, which then executes code on behalf of the VM. This type of context switch requires significant overhead, including saving and restoring the state of both the VM and hypervisor. Additionally, the process of handling a vm-exit can involve multiple steps, such as determining the reason for the exit, emulating hardware behavior, performing necessary translations between the VMs view of the hardware and the physical hardware, and preparing the VMCS for vm-entry (as seen in 2.3.3). These steps can take time, and the overhead associated with them can impact the overall performance of the system, especially if we are doing it for every system call executed in a VM. Although we did not test the performance of the neither the host nor KVM VM, we are certain that the vm-exits causes a significant degradadation in the systems performance.

To minimize the impact of VM-exits on performance, hypervisors often employ techniques like paravirtualization, which allows the virtual machine to interact more directly with the underlying hardware (hypercalls), reducing the need for VM-exits. Additionally, hardware vendors provide features like Intel VT-x or AMD-V, which can accelerate certain hypervisor operations and reduce the overhead associated with VM-exits. However, with Frail, we remove these capabilities when we disable the SYSCALL and SYSRET instructions. The emulation of the SYSCALL instruction is especially resource-intensive for many reasons. Firstly, it requires security checks, such as verifying that the caller has the necessary permissions to execute the SYSCALL. These security checks can add additional processing overhead, which slows down the systems performance. Finally, due to CPU resource contention, causing a vm-exit slows down other virtual machines running on the hypervisor, as well as the host system. This is because the host, and all the running VMs are competing for the same physical resources. Thus, by forcing extra vm-exits, we are delaying the execution of other instructions on other VMs and the host, even though modern CPUs have Out-of-order execution capabilities.

The exact cost of a VM-exit will depend on these factors and can vary widely depending on the specific use case and workload. However, in general, the cost of a VM-exit is considered to be relatively high compared to regular system calls or other operations, which is why minimizing the number of vm-exits is an important consideration when optimizing virtualized environments.

7.2 Hypervisor-based VMI or IPS is Intrusive

Virtual machine introspection (VMI) can be considered intrusive because it involves accessing and analyzing the memory and state of a running virtual machine from the outside, without the knowledge or consent of the guest operating system or applications running inside the virtual machine. This can potentially impact the stability and performance of the virtual machine, as well as introduce security risks and vulnerabilities. For example, if an attacker gains access to the host operating system and uses VMI to access the memory of a virtual machine, they may be able to extract sensitive information, modify the behavior of the virtual machine, or launch attacks against other virtual machines or systems on the same host. Moreover, VMI may also require the use of privileged or kernel-level access to the host operating system, which can increase the risk of system crashes or other adverse effects. It is therefore important to use VMI tools and techniques carefully and judiciously, and to take appropriate precautions to ensure the security and stability of the virtual environment.

Future Work

8.1 Creating a GUI for Frail

If Frail is to be a truly adoptable security solution, it first needs to be a usable one. In its current incarnation, Frail is not user friendly; it supports minimal interaction through a set of simple CLI programs and most user feedback and notification occurs through log files. In order for Frail to become usable software, the daemon needs a graphical user interface to act as a dashboard for user interaction. I envision the Frail GUI as a way for the user to get detailed information about the behavior of their system and make administrative decisions therein. For example, the user could view a visual representation of recent anomalous behavior, inspect profiles for detailed information, make modifications to profiles, and perform operations on running processes such as killing them or stopping a VM that is detected as anomalous. This will allow the user to use Frail as a tool for visualizing system behavior and make easy modifications to Frail's enforcement on the system. Both during and after the development of the GUI, I plan to conduct a usability studies to get an idea of how users interact with Frail, what their perception of the system is, and whether changes need to be made to increase its potential for future adoption.

8.2 Collecting Data from Task_Struct Struct

One of Frail's primary strengths is the ability to monitor the VM from the hypervisor layer. However, getting relevant information from the hypervisor is not trivial due to its semantic gap (as mentioned in 2.9.2). So far, we have been able to trace system calls and its corresponding process name information by reading specific VCPU registers and the CR3 register during a precise context switch. In the future, I would like to obtain process information through the task_struct struct. The task_struct struct is the main data structure in the Linux kernel that represents a process. It contains sensitive information about the process, such as its memory mappings, file descriptors, and credentials. However, it is not trivial to read the task_struct struct from the hypervisor for many reasons. First of all, the task_struct is randomized. Randomizing the task_struct struct is one of the security measures used by the Linux kernel to prevent certain types of attacks, such as stack-based buffer overflows and format string vulnerabilities. When the task_struct struct is randomized, the layout of its fields is changed at runtime, making it more difficult for an attacker to exploit a vulnerability in the kernel. By randomizing the task_struct struct, the kernel ensures that an attacker cannot rely on the knowledge of a fixed offset to access sensitive data in the struct. This technique is commonly referred to as "kernel address space layout randomization" (KASLR), and it is one of the many security measures used by modern operating systems to protect against various types of attacks.

8.2.1 Finding the Per-CPU variable current_task

The current_task is a Per-CPU variable of type task_struct. With a Per-CPU variable, each VCPU in the system will have its own copy of the variable. The current_task variable is used in the Linux kernel to hold a pointer to the currently executing task, also known as the current process or thread. The CPU makes use of this variable to keep track of which task is currently executing on the processor. The current_task variable is a global variable that is accessible from anywhere in the kernel. Therefore, it is also accessible from the hypervisor. If we can somehow get the address of the current_task, then we can read the subsequent 1000 bytes starting from the address of current_task to find more information about a particular process.

8.3 Workload Characterization

Workload characterization is important for VMs because it helps to understand the resource requirements and behavior of the applications running on them. By characterizing workloads, we can gain insights into how different applications use CPU, memory, and disk resources over time. This information can then be used to optimize the allocation of resources to different virtual machines and ensure that they are running efficiently. In a virtualized environment, multiple virtual machines run on the same physical machine or server, sharing its resources. Workload characterization helps in identifying the resource requirements of different VMs and ensures that they are allocated the appropriate amount of resources. This, in turn, helps to prevent resource contention and improve the overall performance of the virtualized environment. Furthermore, workload characterization can also help in capacity planning and forecasting. By understanding the resource utilization patterns of different applications, we can make informed decisions about when to scale up or down resources to meet changing demands. Overall, workload characterization is a critical step in optimizing the performance of VMs and ensuring that they are running efficiently.

Conclusion

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. Master's 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] Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of computer security*, 6(3):151–180, 1998.

[15] Hajime Inoue and Anil Somayaji. Lookahead pairs and full sequences: a tale of two anomaly detection methods. In *Proceedings of the 2nd Annual Symposium on Information Assurance*, pages 9–19. Citeseer, 2007.

[16] Intel. Intel virtualization technology list. In *Intel® Virtualization Technology List*, Unknown.

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

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

[19] 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.

[20] 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.

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

[22] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[23] Kara Nance, Matt Bishop, and Brian Hay. Virtual machine introspection: Observation or interference? *IEEE Security & Privacy*, 6(5):32–37, 2008.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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

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

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

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

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

[34] 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.

Appendix

10.1 Source Code of VM_Exit Counter

```
import os
from bcc import BPF
from time import time

end = time() + 3600
vm_exit_zero_counter = 0
vm_exit_thirty_two_counter = 0

def enable_tracepoint():
    dir_to_tracepoint = [
                      "/sys/kernel",
                      "/sys/kernel/tracing",
                      "/sys/kernel/tracing/events",
                      "/sys/kernel/tracing/events/kvm",
                      "/sys/kernel/tracing/events/kvm/kvm_exit",
                      "/sys/kernel/tracing/events/kvm/kvm_exit/enable"
                  ]

    # this assumes the uid is root/0
    for path in dir_to_tracepoint:
        os.chmod(path, 0o777)

    # open and write '1' to file to enable tracepoint
    enable_kvm_syscall = open("/sys/kernel/tracing/events/kvm/kvm_exit/enable", "w+")
    enabled = int(enable_kvm_syscall.read(1))
    if not enabled:
        enable_kvm_syscall.write("1")
        enable_kvm_syscall.close()



def vm_exit():
    global vm_exit_zero_counter
    global vm_exit_thirty_two_counter
    enable_tracepoint()

    trace = BPF(text="""""", cflags=["-Wno-macro-redefined"])


    while 1 and time() < end:
        try:
            (task, pid, cpu, flags, ts, msg) = trace.trace_fields()
            kvm_exit = msg.decode(errors='ignore').split()
            if kvm_exit[3] == "EXCEPTION_NMI":
                vm_exit_zero_counter += 1
            if kvm_exit[3] == "MSR_WRITE":
                vm_exit_thirty_two_counter += 1
        except ValueError:
            continue

    print("VM EXIT ZERO: " + str(vm_exit_zero_counter))
    print("VM EXIT MSR WRITE: " + str(vm_exit_thirty_two_counter))

vm_exit()
```

Listing 10.1: Source Code of VM-EXIT Counter

```
TRACE_EVENT(kvm_syscall,
    TP_PROTO(int pid, unsigned long cr3, unsigned long vector, int vcpu_number, char*
        process),
    TP_ARGS(pid, cr3, vector, vcpu_number, process),
    TP_STRUCT__entry(
        __field(int, pid)
        __field(unsigned long, cr3)
        __field(unsigned long, vector)
        __field(int, vcpu_number)
        __field(char*, process)
    ),

    TP_fast_assign(
        __entry->pid = pid;
        __entry->cr3 = cr3;
        __entry->vector = vector;
        __entry->vcpu_number = vcpu_number;
        __entry->process = process;
    ),

    TP_printk("pid=%d cr3=%lu vector=%lu vcpu_number=%d process=%s", __entry->pid,
         __entry->cr3, __entry->vector, __entry->vcpu_number, __entry->process)
);
```

Listing 10.2: Implementation of KVM SYSCALL Linux Kernel Custom Tracepoint