

Guest-Based System Call Introspection with Extended Berkeley Packet Filter

by

Huzaiifa 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

© 2022 Huzaiifa Patel

their kindness is masquerade.

yearning to occupy one with false pretenses.

it's used to sedate.

I promise you'll get this when the sky clears for you.

Abstract

Soon

Acknowledgments

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

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

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

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

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

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

Contents

Abstract	i
Acknowledgments	ii
List of Figures	vii
List of Tables	viii
Nomenclature	ix
1 Introduction	1
1.1 The Problem	2
1.2 Addressing the Problem	3
1.3 Research Questions	3
1.4 Motivation	4
1.4.1 Why Design a New VMI?	5
1.4.2 Why Design a Hypervisor-Based VMI System?	6
1.4.2.1 Hypervisor-Based VMI's	6
1.4.2.1.1 Isolation	6
1.4.2.1.2 Inspection	7
1.4.2.1.3 Inspection	8
1.4.2.1.4 Deployability	8
1.4.2.2 Guest-Based VMI's	8
1.4.2.2.1 Privilege Escalation	8
1.4.2.2.2 Tampering	9
1.4.2.2.3 Rich Abstractions	10
1.4.2.2.4 Speed	10

1.4.2.3	Conclusion	11
1.4.3	Why eBPF?	11
1.4.4	Why Utilize System Calls for Introspection?	13
1.4.5	Why Utilize Sequences of System Calls?	14
1.5	Related Work	15
1.5.1	Properties of Nitro	15
1.5.1.1	Guest OS Portability	15
1.5.1.2	Evasion Resistant	16
1.5.2	Implementation	16
1.5.2.1	Nitro Client Side Implementation	16
1.5.2.2	VMI Mechanisms for Tracing System Calls From The Host	16
1.5.2.3	How Nitro Empowers Anomaly Detection	17
1.6	Contributions & Improvements On Related Work	17
1.7	Thesis Organization	18
2	Background	19
2.1	Overview of Hypervisors	19
2.1.1	Type 1 Hypervisor	19
2.1.2	Type 2 Hypervisor	20
2.1.3	Problems With Type 1 & Type 2 Hypervisor Classifications	20
2.1.4	Native Hypervisor	21
2.1.5	Emulation Hypervisor	21
2.2	x86-64 Intel Central Processing Unit	22
2.2.1	Exceptions	22
2.2.2	Faults	22
2.2.3	Traps	23
2.2.4	Instructions	24
2.2.5	Registers	25
2.2.6	%rip Register	25
2.2.7	%rdi Register	25
2.2.8	Protection Rings	26
2.2.9	Execution Modes	28
2.2.10	Model Specific Register (MSR)	28
2.3	Intel Virtualization Extension (VT-X)	31

2.3.1	Overview	31
2.3.2	Novel Instruction Set	33
2.3.3	The Virtual Machine Control Structure (VMCS)	35
2.3.4	VM-Exit	37
2.3.5	VM-Entry	44
2.4	System Calls	46
2.5	The Kernel Virtual Machine (KVM) Hypervisor & QEMU	46
2.6	Virtual Machine Introspection	53
2.7	eBPF	54
2.7.1	Overview	54
2.7.2	How Does eBPF Work?	55
2.8	The Linux Kernel Tracepoint API	55
2.8.1	Overview	55
2.8.2	Identifying Traceable Kernel Subsystems	56
2.8.3	Identifying Tracepoint Events	56
2.8.4	Tracepoint Format File	56
2.8.5	Using Tracepoint Events with eBPF	57
2.9	pH-based Sequences of System Call	57
3	Designing Frail	58
3.1	Tracing KVM VM System Calls	58
3.2	Tracing KVM VM Processes	61
4	Future Work (Winter 2022)	64
4.1	Implementing Frail	64
4.1.1	User Space Component	64
4.1.2	Kernel Space Component	64
4.1.2.1	Custom Linux Kernel Tracepoint	64
4.1.2.2	Kernel Module	64
4.1.3	Tracing Processess	64
4.1.4	Proof of Tracability of all KVM Guest System Calls	64

List of Figures

2.1	Mental Model of Type 1 & Type 2 Hypervisor	20
2.2	Life Cycle of an Exception	24
2.3	High Level Illustration of Instruction Format	25
2.4	Illustration of the Intel x86 Protection Ring	28
2.5	Representation of the IA32_EFER MSR (0xC0000080)	31
2.6	Illustration of VMX Root & Non-Root Mode in Relation to Intel Protection Rings. . . .	33
2.7	Life Cycle of a VM-Exit on invalid opcode	44
2.8	Life Cycle of a VM-Entry	45
2.9	Successful Hypervisor Life Cycle Under Intel VMX	46
2.10	Decision on Whether QEMU use TCG or CPU for Executing an Arbitrary Instruction X. . . .	49
2.11	Partial KVM Life Cycle if TCG is Disabled	53
3.1	Illustration of Tracing KVM VM System Call	61
3.2	Illustration of Tracing KVM Guest Processes	63

List of Tables

2.1	IA32_EFER MSR (0xC0000080)	30
2.2	Instructions that could cause conditional VM-exits as defined by the VM-exit control section of the VMCS	39
2.3	Intel VMX Defined VM-Exits	40

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

Introduction

Cloud computing is a modern method for delivering computing power, storage services, databases, networking, analytics, artificial intelligence, 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, email, virtual desktops, software development, testing, big data analytics, and web applications [2]. For example, healthcare companies are using the cloud to store patient records in databases [2]. Financial service companies are using the cloud for real-time fraud detection and prevention [2]. And finally, video game companies are using the cloud to deliver online video game services to millions of players around the world.

The existence of cloud computing can be attributed to virtualization. Virtualization 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. Virtualization 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 (CPU), 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 hypervisors resources is called the guest OS.

While virtualization technology can be sourced back to the 1970s, it wasn't widely adopted until the early 2000s due to hardware limitations [16]. The fundamental rea-

son for introducing a hypervisor layer on a modern machine is that without one, only one operating system would be able to run at a given time. This constraint often led to wasted resources, as a single OS infrequently utilized a modern hardware's full capacity. 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 utilized 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

Due to exposure to the Internet, VMs represent a first point-of-target for attackers who want to gain access into the virtualization environment [20]. A VM that is exposed to the Internet is changing constantly due to influx of non-deterministic stream of data coming from the Internet and into the VM [17]. Apart from the Internet, another problem is the simple fact that modern day computer systems run dozens, if not hundreds of programs that each contain a remarkable amount of complexity and functionality [17]. The required capabilities and complexity of both computer programs and the system has led to a reduction in their reliability and security [17]. For instance, new vulnerabilities are discovered almost every day on the majority of major computer platforms. When these vulnerabilities are addressed with software updates, it is not uncommon for new vulnerabilities to be discovered [17]. As such, the role of a VM is highly security critical and its priority should be to maintain confidentiality, integrity, authorization, availability, and accountability throughout its existence [18]. The successful exploitation of a VM can result in a complete breach of isolation between clients, resulting in the failure to meet one or more of the aforementioned priorities of computer security. For example, the successful exploitation of a VM can result in the loss of availability of client services due to denial-of-service attacks, non-public information becoming accessible to unauthorized parties, data, software or hardware being altered by unauthorized parties, and the successful repudiation of a malicious action committed by a principal [18]. For these reasons, effective methodologies for monitoring VMs is required.

1.2 Addressing the Problem

In this thesis, we present Frail, a KVM hypervisor and Intel VT-x exclusive hypervisor-based virtual machine introspection (VMI) system that enhances the capabilities of Nitro, which is a related VMI system. In computing, VMI is a technique for monitoring and sometimes responding to the runtime state of a virtual machine [13]. Frail is a VMI that (1) traces KVM guest system calls, (2) monitors malicious anomalies, and (3) responds to those malicious anomalies from the hypervisor level. Our framework is implemented using a combination of existing software and our own software. Firstly, it utilizes Extended Berkeley Packet Filter (eBPF) to safely extract both KVM guest system calls and the corresponding process that requested the system call. Secondly, it uses Dr. Somayaji’s pH [17] implementation of sequences of system calls to detect malicious anomalies. Lastly, we make use of our own software to respond to the observed malicious anomalies by slowing down or terminating the guest process that is responsible for the observed malicious anomaly. The tracing, monitoring, and responding is done in real-time without hindering usability of the guest. To our knowledge, Frail is the second hypervisor-based VMI system that is intended to support the monitoring of all three system call mechanisms provided by the Intel x86 architecture, and has been proven to work for Linux 64-bit systems. Likewise, it is the first KVM hypervisor-based VMI system that utilizes sequences of system calls to monitor for malicious anomalies.

1.3 Research Questions

In this thesis, we consider the following research questions:

Research Question 1: KVM is formally defined as a type 1 hypervisor. As a result, guest instructions interact directly to the CPU. Can we change the route of system calls so that they are trapped and emulated at the hypervisor level?

Research Question 2: Can we effectively retrieve KVM guest system calls and the corresponding 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 malicious anomalies in real-time with a high success rate, and without hindering the usability of the guest?

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

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

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

1.4 Motivation

Current Linux computer systems do not have a native general-purpose mechanism for detecting and responding to malicious anomalies within KVM VMs. As our computer systems grow increasingly complex, so too does it become more difficult to gauge precisely what they are doing at any given moment. Modern computers are often running dozens, if not hundreds of processes at any given time, the vast majority of which are running silently in the background. As a result, users often have a very limited notion of what exactly is happening on their systems, especially beyond that which they can actually see on their screens. An unfortunate corollary to this observation is that users also have no way of knowing whether their system may be misbehaving at a given moment. For this reason, we cannot rely on users to detect and respond to malicious anomalies. If users are not good candidates for adequately monitoring our VMs for malicious anomalies, computer systems should be programmed to watch over themselves through the hypervisor. Due to VMs being highly security critical, we have turned to VMI to provide the necessary tools to help trace, monitor and respond to malicious

anomalies found within KVM VMs. What follows is a comprehensive explanation into our motivation for designing our VMI in the manner that we did.

1.4.1 Why Design a New VMI?

The topic of securing virtual machines (VMs) dates back to 2003, when Tal Garfinkel and Mendel Rosenblum proposed VMI as a hypervisor-level intrusion detection system (IDS) that integrated the benefits of both network-based and host-based IDS [10] [17]. 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 framework, 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 named Nitro that is similar to our VMI. More specifically, Nitro is a VMI for system call tracing and monitoring, which was intended, implemented, and proven to support Windows, Linux, 32-bit, and 64-bit environments [15]. 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 compatible with newer Windows desktop versions. In fact, at the time of writing this thesis, Nitro only supports Windows XP x64 and Windows 7 x64, which makes Nitro entirely ineffective for two reasons. Firstly, both Windows XP and Windows 7 is a discontinued OS, which means that security updates and technical support are no longer available. Secondly, at the time of writing, Windows XP is now over 21 years old and consists of only 0.39% of the marketshare of worldwide Windows desktop versions running [17]. Similarly, Windows 7 is 13 years old, and consists of only 9.6% of the marketshare of worldwide Windows desktop versions running [17].

There is a fundamental problem with the state of many existing VMI's like Nitro: when the codebase of either an OS or the kernel changes, VMI's are unable to solve the problem for which they were originally designed to solve - to trace and monitor VMs that are running Windows, Linux, 32-bit, and 64-bit environments [20]. The primary reason for problem is that VMIs were designed in such a way that compromised com-

patibility 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 issues with VMI's like Nitro, we seek to design a spiritual successor to Nitro that is intended to provide a VMI without sacrificing compatibility with subsequent versions of the Linux kernel. We will extensively discuss how we intend to accomplish this the "Contributions" section and "Implementation" chapter.

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 monitoring), or it can be placed on the hypervisor level outside of any VM (Hypervisor-based VMI). In this section, we justify our motivations for designing and implementing a hypervisor-based VMI by analyzing the advantages and disadvantages of both hypervisor-based and guest-based VMI's.

1.4.2.1 Hypervisor-Based VMI's

Hypervisor-based VMIs offer four key advantages over traditional guest-based VMI's: (1) isolation, (2) inspection, (3) interposition, and (4) deployability [14].

1.4.2.1.1 Isolation

In our context, isolation refers to the property that hypervisor-based VMIs are tamper-resistant from its VMs. Tamper resistant in our context is the property that VMs are unable to commit unauthorized access or altering of any of the components of the hypervisor (i.e. code, stored data, and more). First, if we assume that a hypervisor is free of vulnerabilities, then the hypervisor-based VMI is considered isolated from every guest. This implication holds true because hypervisor-based VMIs run at a higher privilege level than guests [10]. It is important to note that guest-based VMs cannot hold the property of isolation due to being deployment within the guest.

When the property of isolation holds for a hypervisor-based VMI, there exists two key advantages:

Firstly, if a hypervisor manages 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 corresponding confined environment. If a hypervisor-based VMI holds the property of isolation, then both the VMI and hypervisor will be immune from attacks that originate in the guest, even if the VMI is actively monitoring a guest that has been attacked [10].

The second advantage is that 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 (guest-based VMI), the entire guest Linux kernel would need to be trusted. Having to trust only the hypervisor is advantageous 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's. Although attackers may still be able to generate false data by tampering the guest, the hypervisor-based VMI is guaranteed to be safe. If required, the VMI could also extend its capabilities to successfully defend against false guest data generation attacks.

1.4.2.1.2 Inspection

Inspection refers to the property that allows the VMI to examine the entire state of the guest while continuing to be isolated [14]. 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 efficiently having a complete view of all guest OS states (CPU registers, memory, devices, disk state, and more) [10]. For example, we can observe each processes state, as well as the kernel state, including those hidden by attackers, which is often challenging to achieve through guest-based VMI. 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 would stop executing when a VM goes into a paused state.

1.4.2.1.3 Inspection

Interposition is the 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, the VMI is capable of modifying any of the states of the guest and interfering with every activity of the guest. With respect to our VMI, interposition makes it easy to respond to observed malicious anomalies by slowing down the guest process responsible for the malicious anomaly [10].

1.4.2.1.4 Deployability

Deployability of a VMI refers to the ease with which it can be taken from development to deployment onto 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 hypervisor-based VMI at the hypervisor layer, no guest has to be modified to accomodate for the VMI's deployment. For example, we do not have to make a user for any guest, we do not need to install the VMI software in any of the guests, and we do not have to install any of the VMI's dependencies inside any of the guests. Instead, we only need to install dependencies of the VMI on the host once. Afterwards, we may execute our VMI on the host without disrupting any services in the host or guest.

1.4.2.2 Guest-Based VMI's

Although guest-based VMI systems have been successful, they are more susceptible to two types of threats: (1) privilege escalation, and (2) tampering [14].

1.4.2.2.1 Privilege Escalation

Unlike hypervisor-based VMI's, guest-based VMI's are not isolated because they are executed on the same privilege level as the VM(s) that they are protecting [1]. 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, a design flaw, or a configuration oversight 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. Attackers can carry out unauthorised actions with these additional privileges. For instance, if an attacker successfully escalates their privilege, they can gain access to VMI resources that would normally be restricted to them.

1.4.2.2.2 Tampering

Assuming that our VMI is a guest-based hypervisor, if an attacker successfully escalates their privilege in the guest, the following scenarios are possible:

- An attacker can tamper with the tracing software that collects system call information and/or process/task information that requested the system call.
- 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 software that handles sequences of system calls, which is the tool that monitors for anomalous system calls. In this scenario, attackers can prevent anomalous system calls from being declared.
- The software that responds to processes that requested anomalous system calls can be tampered with. Currently, our security policy consists of either slowing down or terminating the anomalous process. Attackers can tamper our security

policy so that the process that requested the anomalous system call is never slowed down or terminated.

- The database/log files that contains information about anomalous system calls and process information can be tampered with by overwriting or appending them with false data.
- As our VMI is deployed using a kernel module, attackers with escalated privilege can simply remove or shut down the kernel module or process to stop the VMI.

In all the above cases, As long as an attack results in the VMI to continue its normal execution (e.g., no crashes), the VMI system can successfully generate a false pretense to mislead the VMI that a VM state is not malicious, when in fact it is.

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

1.4.2.2.3 Rich Abstractions

With guest-based VMI's, we are able to trivially intercept system calls and process information due to the user space interfaces provided to extract OS level information. We can use critical kernel variables and functions to trace system call and process information. Or, even simpler, we can also use the available third party Linux tools like strace to extract system calls inspect their arguments, return values, or sequences. We can also use the /proc directory to obtain process information.

1.4.2.2.4 Speed

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. The most effective way optimize a VM is to reduce the number of VM-Exits [11]. We discuss about hypervisor traps further in the "Background" chapter.

1.4.2.3 Conclusion

We believe that the disadvantages of guest-based VMI's outweigh its advantages. More specifically, the security of both our VMI and the VM's that require monitoring are more important than rich abstractions and speed that guest-based VMI's provide. For that reason, we have designed and implemented a hypervisor-based VMI.

1.4.3 Why eBPF?

As previously mentioned, most organizations today use cloud-computing environments and virtualization technology. In fact, Linux-based clouds are the most popular cloud environments among organizations, and thus have become the target of cyber attacks launched by sophisticated malware [12]. As a result, security experts, and knowledgeable users are required to monitor systems with the intent of maintaining the goals of computer security. The demand for monitoring 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 many tracing softwares. What follows is an explanation of why we selected eBPF to perform the tracing and monitoring of KVM guest system calls and the corresponding process that requested the system call.

Historically, due to the kernel's privileged ability to oversee and control the entire system, the kernel 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 modifying 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 syntactical error in a user space application, it could crash the corresponding user space process. Likewise, if there exists a logical or syntactical 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. The purpose of a VMI is to debug or conduct forensic analysis on a VM [13]. If the implementation of the VMI system hinders that purpose, it would become an ineffective VMI. To limit the amount of Linux kernel modifications and kernel module insertions required to implement our

VMI, we chose to use eBPF to trace and monitor KVM guest system calls and the corresponding process that requested the system call. This is due to two 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 advantages of eBPF extend far beyond scope of traceability; eBPF is also extremely performant, and runs with guaranteed safety. In practice, this means that eBPF is an ideal tool for use in production environments and at scale. Safety is guaranteed with the help of a kernel space verifier that checks all submitted bytecode before its insertion into the eBPF VM. For example, the eBPF verifier analyzes the program, asserting that it conforms to a number of safety requirements, such as program termination, memory safety, and read-only access to kernel data structures. For this reason, eBPF programs are far less likely to adversely impact a production system than other methods of extending the kernel (e.g. modifying the 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 communicates with user space via both predefined and custom Linux kernel tracepoints. As a result, the number of context switches required between the user space and kernel space is greatly diminished.

Trust and support in eBPF has found its way into the infrastructure software layer of giant data centers. For instance, eBPF is already being used in production at large data-centers by Facebook, Netflix, Google, and other companies to monitor server workloads for security and performance regressions [64]. Facebook has released its eBPF-based load balancer Katran which has been powering Facebook data centers for several years now. eBPF has long found its way into enterprises. Examples include Capital One and Adobe, who both leverage eBPF via the Cilium project to drive their networking,

security, and observability needs in cloud-native Kubernetes environments. eBPF has even matured to the point that Google has decided to bring eBPF to its managed Kubernetes products GKE and Anthos as the new networking, security, and observability layer. The trust in eBPF by big companies has incentivized us and factored into our decision to make a VMI that utilizes eBPF.

In summary, eBPF offers unique and promising advantages for developing novel security mechanisms. Its lightweight execution model coupled with the flexibility to monitor and aggregate events across userspace and kernelspace provide the ability to control and audit every KVM guest system call. eBPF maps, shareable across programs and between userspace and the kernel offer a means of aggregating data from multiple sources at runtime and using it to inform policy decisions like slowing down or terminating a malicious process caught by KVM sequences of system calls. A VMI partially implemented with eBPF can be dynamically loaded into the kernel as needed, and eBPF's safety guarantees combined with it being a native Linux technology provides strong adoptability advantages. This means that a VMI based on eBPF can be both adoptable and effective.

1.4.4 Why Utilize System Calls for Introspection?

One of the design decisions that are considered when implementing a hypervisor-based VMI system is by asking the following question: What Linux system event can be traced and monitored to identify the presence of a malicious anomaly within a system, with a high success rate and a low false positive/negative rate? Existing research in VMI systems have answered the foregoing question by successfully utilizing guest system call as their target event from the hypervisor level, and proving its effectiveness in relation to performance and functionality by providing extensive test results with various guest OSs. As a result, we have chosen to utilize system calls events in our VMI system. What follows is high-level definition explanation of what a system call is, and an explanation of why the system call interface has several special properties that make it a good choice for monitoring program behavior for security violations.

On UNIX and UNIX-like systems, user programs do not have direct access to hard-

ware resources; instead, one program, called the kernel, runs with full access to the hardware, and regular programs must ask the kernel to perform tasks on their behalf. Running instances of a program are known as processes.

The system call is a request by a process for a service from the kernel. The service is generally something that only the kernel has the privilege to perform. For example, when a process wants additional memory, or when it wants to access the network, disk, or other I/O devices, it requests these resources from the kernel through system calls. Such calls normally takes the form of a software interrupt instruction that switches the processor into a special supervisor mode and invokes the kernel's system call dispatch routine. If a requested system call is allowed, the kernel performs the requested operation and then returns control either to the requesting process or to another process that is ready to run.

Hence, system calls play a very important role in events such as context switching, memory access, page table access and interrupt handling. With the exception of system calls, processes are confined to their own address space. If a process is to damage anything outside of this space, such as other programs, files, or other networked machines, it must do so via the system call interface. Unusual system calls indicate that a process is interacting with the kernel in potentially dangerous ways. Interfering with these calls can help prevent damage, and help maintain the stability and security of a VM. previously created VMI's have utilized system calls to passively flag any unusual, anomalous, or prohibited behavior with a high success rate, without hindering the overall performance of the virtualization environment, and while keeping the guest OS active.

1.4.5 Why Utilize Sequences of System Calls?

A neural network implementation is a modern approach to utilizing sequences of system calls to detect malicious abnormalities in VMs. Although the classic system call sequences implementation of pH requires a less complex implementation than that of a neural network implementation, we believe complexity does not equate to better. Our motivation for using an pH's implementation on system call sequences is because Somyaji proved its effectiveness in his paper. Although the original design is twenty

years old, we believe it is still effective in detecting and responding to malicious processes.

1.5 Related Work

In this chapter, we will take a look at Nitro, a hardware-based VMI system that utilizes guest system calls for the purpose of monitoring and analyzing the state of a virtual machine. Nitro is the first VMI system that supports all three system call mechanisms provided by the Intel x86 architecture, and has once 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 detecting malicious activity within a VM.

1.5.1 Properties of Nitro

1.5.1.1 Guest OS Portability

Guest OS portability refers to a property that allows the same VMI mechanism to work for various guest OSs without major changes. The goal of Nitro's VMI system is to allow any guest OS to work without making any changes in the codebase implementation. To achieve this, the underlying mechanism of Nitro does not rely on the guest OS itself, but rather on the VMs hardware specification. For example, Nitro uses a feature provided by the Intel x86 architecture to trace system calls. Therefore, how system call tracing is possible is specified and defined by the x86 architecture. Therefore, all guest OSs running on this hardware must conform to these specifications. As Nitro is a VMI that intended for the Intel x86 architectures, it uses hardware specific capabilities to allow the guest OS to work on any OS that uses Intel x86 architecture.

1.5.1.2 Evasion Resistant

Nitro provides a mechanism known as hardware rooting to guarantee their VMI is evasion resistant. Hardware rooting is the VMI mechanism that bases its knowledge on information about the virtual hardware architecture, these attacks cannot be applied.

That is, these mechanisms cannot be manipulated in a way which allows a malicious entity to circumvent system call tracing or monitoring.

1.5.2 Implementation

This section describes the implementation of Nitro. Nitro is based on the KVM hypervisor. It is good to note that KVM is split into two portions, namely a host user space application that is built upon QEMU and a set of Linux kernel modules.

1.5.2.1 Nitro Client Side Implementation

The user application portion of KVM provides the QEMU monitor which is a shell-like interface to the hypervisor. It provides general control over the VM. For example, it is possible to pause and resume the VM as well as to read out CPU registers using the QEMU monitor. Nitro modified KVM by adding new commands to the QEMU monitor to control Nitro's features. That is, all Nitro commands are input via the QEMU monitor. These commands are then sent to the kernel module portion of KVM through an I/O control interface.

1.5.2.2 VMI Mechanisms for Tracing System Calls From The Host

When Nitro was implemented, trapping to the hypervisor on the event of a system call was not supported on Intel IA-32 (i.e. x86) and Intel 64 (formerly EM64T) architectures. As a result, Nitro found a way to indirectly trap to the hypervisor in the event of a system call. Nitro does this by forcing system interrupts (e.g. page faults, general protection faults (GPF), etc) for which trapping is supported by the Intel Virtualization Extensions (VT-x). Since there are three system call mechanisms defined by the x86 architecture, and because they are quite different in their nature, a unique trapping

mechanism was designed for each.

1.5.2.3 How Nitro Empowers Anomaly Detection

Nitro’s implementation allows for tracing KVM guest system calls From the host. However, Nitro does not monitor for anomalous systems, nor does it respond to anamolous system calls. Instead, Nitro expects external applications to utilize Nitro’s system call tracing capabilities to perform the monitoring and responding of anomalous system calls. Different applications for system call monitoring want a varying amounts of information. In some cases an application may want only a simple sequence of system call numbers, while other application may require detailed information including register values, stack-based arguments, and return values from a small subset of system calls. As Nitro cannot foresee every need of applications that conduct system call monitoring and responding, Nitro does not deliver a fixed set of data per system call. Instead, it allows the user to define flexible rules to control the data collection during system call tracing. Based on the user specification, Nitro will then extract the system call number. It is always important to be able to determine which process produced a system call. Therefore, Nitro will also extract the process identifier. With these capabilities, Nitro can be used effectively in a variety of applications, such as machine learning approaches to malware detection, honeypot monitoring, as well as sandboxing environments.

1.6 Contributions & Improvements On Related Work

To summarize, our contributions are as follows:

- Nitro’s implementation only allows tracing of system calls of KVM VMs that are created with QEMU. Our VMI provides the ability to trace every KVM guest system call and and their corressponding guest process no matter how the KVM VM was created.
- We extend the Linux kernel tracepoint API in the host OS to define two new events: (1) KVM guest system calls and (2) guest processess that requested a

system call. The API extension allows eBPF programs to instrument these two events.

- Nitro is not capable of monitoring and responding to anomalous KVM guest system calls. With our prototype, we provide the ability to monitor and respond to anomalous KVM guest system calls by triggering the hypervisor to satisfy a variety of security policies. More specifically, our monitoring of anomalous system calls will be done in real time with pH. And our VMI's response system will be able to effectively delay or terminate an anomalous KVM guest process. Essentially, we are including an intrusion detection system into our VMI.

1.7 Thesis Organization

The rest of this thesis proceeds as follows:

- Chapter 2: We present detailed a background information on VMI systems, virtualization, system calls, the Linux kernel, the Linux tracepoint API, and eBPF.
- Chapter 3: We take a look at the design of our VMI.
- Chapter 4: We take a look at the implementation of our VMI.
- Chapter 5: We hypothesize the result of our VMI based on our design and implementation.
- Chapter 6: We explore our plan of action for the second term.

Background

This chapter presents technical background information required to understand this thesis and discusses related work from the perspective of industry and academia.

Section 2.1 provides the different definitions of hypervisors. Section 2.2 explains the Intel Virtualization Extension (VT-x), which we utilize in our VMI prototype. Section 2.3 explains how the KVM hypervisor works. Section 2.4 explains the relationship between Quick Emulator (QEMU) and KVM. Section 2.5 comprehensively explains how a system call works in Linux systems. Section 2.6 provides the definition of a VMI.

2.1 Overview of Hypervisors

As previously mentioned, a hypervisor is a type of computer software that allows virtual machines to be created and ran on a machine. Depending on where on the machine the hypervisor is located, hypervisors can be classified into two types: (1) type 1 hypervisor and (2) type 2 hypervisor.

2.1.1 Type 1 Hypervisor

Type 1 hypervisors run directly on physical hardware to create, control, and manage VMs. Type 1 hypervisors do not require require the host OS. Instead, they have their own drivers. Type 1 hypervisors are also called native or bare-metal hypervisors. The first hypervisors, which IBM developed in the 1960s, were native hypervisors. [11]. Examples of type 1 hypervisors include, but are not limited to Xen, VMware ESX, Microsoft Hyper-V.

2.1.2 Type 2 Hypervisor

Type 2 hypervisors consists of installing the hypervisor on top of the actual operating system (Windows, Linux, MacOS), just as other computer programs do. In other words, a type 2 hypervisor runs as a process on the host OS. Type-2 hypervisors abstract guest operating systems from the host operating system by becoming a third software layer above the hardware, as shown in figure 2.1. Type 2 hypervisors are also called hosted hypervisors. Examples of type 2 hypervisors include but are not limited to KVM, VMware Workstation, VirtualBox, and QEMU.

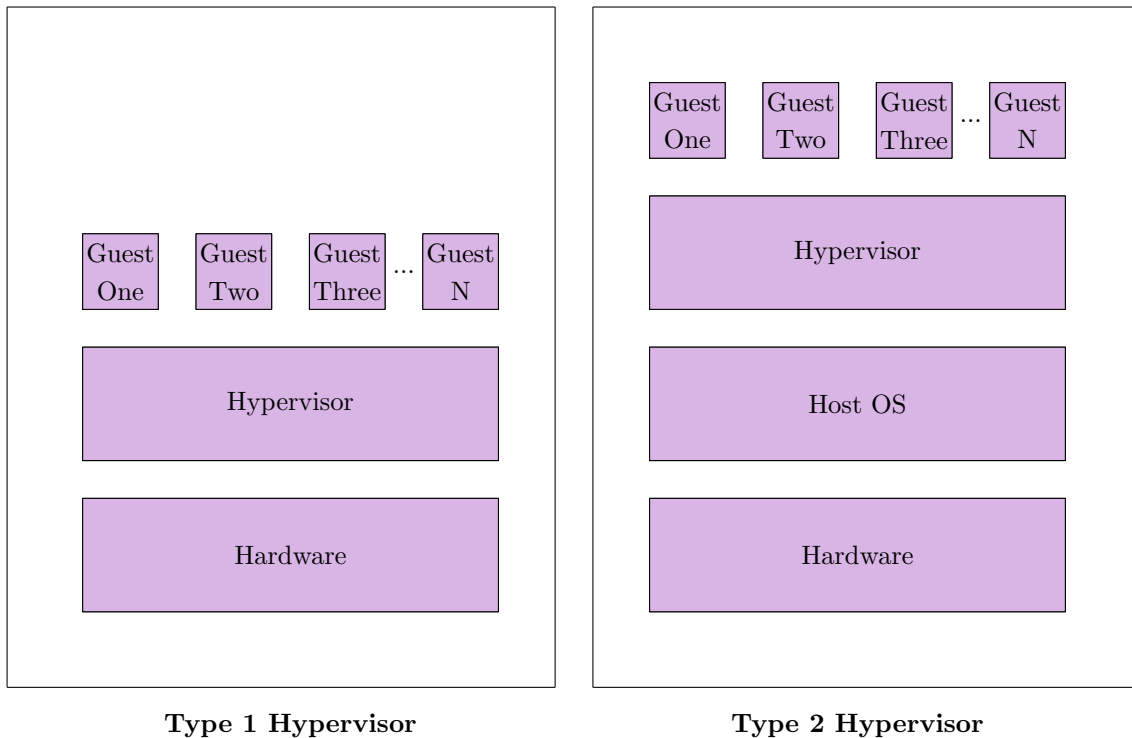


Figure 2.1: Mental Model of Type 1 & Type 2 Hypervisor

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, KVM

is implemented and deployed using two Linux kernel modules that effectively convert the host operating system into a type-1 hypervisor according to its creator RedHat [9]. At the same time, KVM can be categorized as a type 2 hypervisor because the host OS is still fully functional and 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 [21].

Due to disagreements and vagueness in the classification of some hypervisors, a new type of classification of hypervisors was defined with the intent to clarify the ambiguity that the type 1 and type 2 definitions has caused [1]. With the new definitions, hypervisors can be classified into two types: (1) native hypervisors and (2) emulation hypervisors [1].

2.1.4 Native Hypervisor

Native hypervisors are hypervisors that push VM guest code directly to the hardware using hardware virtualization extensions like Intel VT-x. We will write about Intel VT-x in the next section [1]. Examples of Native hypervisors include but are not limited to Xen, KVM, VMware ESX, and Microsoft HyperV.

2.1.5 Emulation Hypervisor

Emulation hypervisors are hypervisors that emulate every VM guest instruction using software virtualization [1]. Emulated guest instructions very easy to trace because all instructions can be conveniently trapped to the hypervisor. Examples of emulation hypervisors include but are not limited to QEMU, Bochs, and early versions of VMware-Workstation and VirtualBox [1].

2.2 x86-64 Intel Central Processing Unit

2.2.1 Exceptions

Exceptions are type of signals sent from a hardware device or user space process. to the CPU, telling it to immediately stop whatever it is currently doing either due to an abnormal, unprecedented, or deliberate event that occurred during the execution of a program. When a user space process causes an exception, the control is transitioned from user mode (ring 3) to kernel mode (ring 0). When this happens, the values of all the CPU registers of the process that caused the exception will be saved to memory, so that the process can be loaded again in the future. After this, the kernel will 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 [34]. 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 transferred back to the user space process. It is worth noting that all of the previously mentioned steps are dependent on the CPU scheduler. For example, even if an exception is handled successfully, the CPU may choose to resume another user space process first 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 design and implementation of our VMI. Faults and traps are the only exceptions that are utilized by our VMI. Therefore, we will not go into detail about aborts.

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 [6]. There exists many types of faults, which are each executed for different reasons. How-

ever, we will only introduce the Invalid Opcode (`#UD`) exception due to its relevance to our thesis. A `#UD` exception, also called an undefined instruction is a fault that is generated when an instruction that is sent to a CPU is undefined (not supported) by the CPU. Some faults can be corrected (with kernel function handlers) such that the program that caused the fault may continue as if nothing happened. However, if a fault, such as a `#UD` exception cannot be handled successfully by a relevant kernel function handler, then the computer will halt, and will in some cases require a reboot.

2.2.3 Traps

A trap is a type of exception that is solely triggered by a user space process. When the OS detects a trap, it pauses the user process, and executes the relevant trap handler inside the kernel. There exists different types of traps, which are each executed for different reasons. However, we will only introduce the single stepping trap. Single stepping is a mechanism that the Intel x86 CPU architecture provides. Its purpose is to generate a trap after executing an instruction. As long as single stepping is enabled, every instruction will trap to kernel space. Any program can activate single stepping by using an existing 3rd party software like GNU Debugger (GDB). When single stepping is enabled, there is no need to put a breakpoint/trap to a specific line of code, because every instruction will cause a trap.

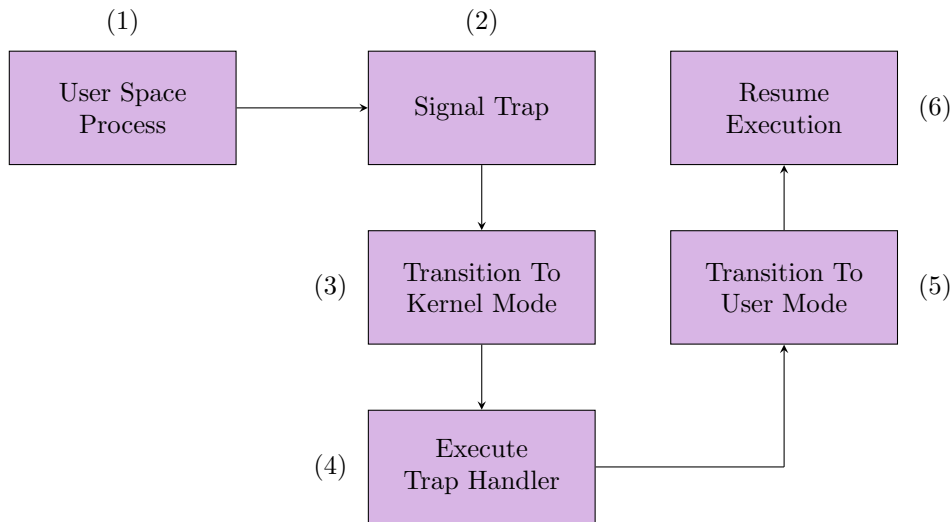


Figure 2.2: Life Cycle of an Exception

2.2.4 Instructions

In this subsection, we discuss briefly and in a high level what CPU instructions are with the intent to aid in understanding the design of our VML.

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. 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, either the SYSCALL, SYSENTER, or SYSRET operation will be processed by the CPU when a process executes a system call. Informally, (3), (4), and (5) 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. Firstly, an operand can be stored in the register. This is known as direct operand. (2)

Or, the address of the operand can be stored in the register. This is called an indirect operand.

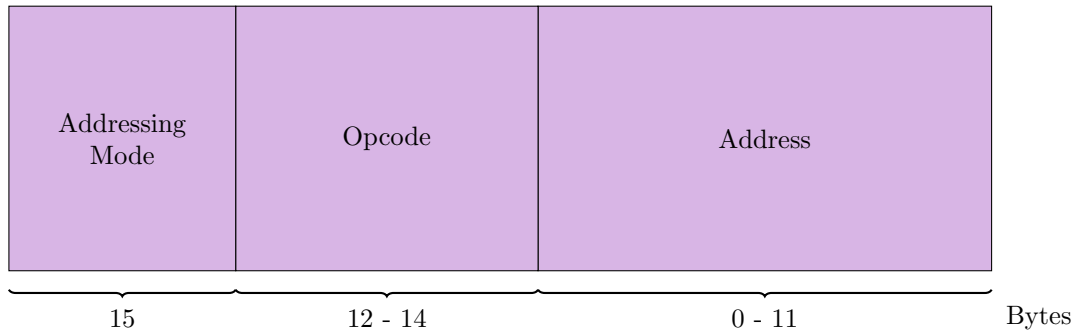


Figure 2.3: High Level Illustration of Instruction Format

2.2.5 Registers

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

2.2.6 `%rip` Register

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

2.2.7 `%rdi` Register

when we call a function, we have to choose some registers to hold the function 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.

2.2.8 Protection Rings

Before we explore the hypervisor further, we must introduce protection rings (also known as privilege 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 (that did not generate a fault). For example, a process that generated a fault would crash itself, but would also cause a perfectly running process to crash with it [19]. Due to these problems, protection 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 OS's 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 describe protection rings as a hierarchical system that consists of four layers: Ring 0, Ring 1, Ring 2, and Ring 3. Next, we describe how portions of the OS are separated into each of these four rings.

First, the OS and all of its processes, functions, user applications, etc., are appointed to a specific ring. This ring is the only place where these processes are permitted to execute. If a process in one ring needs another process or resources from another ring, it must conform to the following directive:

Communication between each ring are strictly controlled. Each layer only works with the layer above/below it. As an example, Ring 3 can only communicate with Ring 2. Ring 2 can communicate with Ring 1 and Ring 3, but not Ring 0.

Ring 0 is where the operating system kernel resides and runs. This ring has the highest level of privileges. The kernel resides in ring 0 because it is responsible for providing services for all other parts of the OS. This level of permission is referred to as kernel, privileged, and/or supervisor mode. In this mode, privileged instructions are executed and protected areas of memory may be accessed [19].

Linux kernel Ring 1 is typically where other OS components that are not in the

kernel run. This ring also runs in privileged mode. Ring 2 is where software-like device drivers run. Currently, ring 1 and 2 are usually unused by most OSes for four reasons: (1) Intel x86 is the only notable architecture that supports ring 1 and ring 2, (2) paging doesn't differentiate between rings 1, 2 and 3, (3) the introduction of Intel VT-x stopped hypervisors from running in Rings 1 and 2, and (4) rings 1 and ring 2 were initially designed to separate privileged drivers from actual kernel code but quickly abandoned because it's more work than it's worth.

Ring 3 is where user applications and programs run. This ring has the least amount of privileges and permissions, and is said to run in user mode. In user mode, the executing code has no ability to directly access hardware or reference memory. Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always recoverable. Most of the code running on your computer will execute in user mode. As such, when certain user space process instructions require processes or resources from more privileged rings, the user application will issue a system call to the next ring in order to obtain the appropriate service.

The segmentation that protection rings creates, 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.

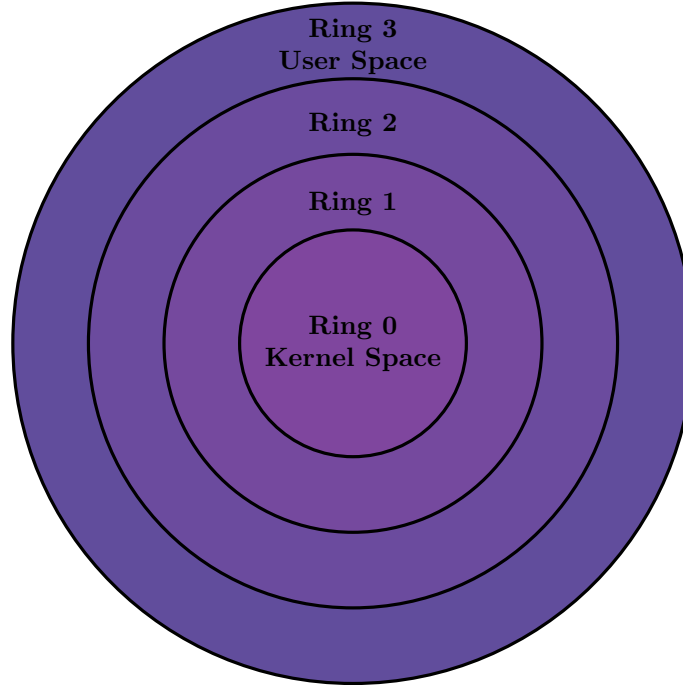


Figure 2.4: Illustration of the Intel x86 Protection Ring

2.2.9 Execution Modes

The x86 has been extended in many ways throughout its history, remaining mostly backwards compatible while adding execution modes and large extensions to the instruction set. A modern x86 processor can operate in one of four major modes: 16-bit real mode, 16-bit protected mode, 32-bit protected mode, and 64-bit long mode.

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 i386 CPUs, Intel implemented two model specific registers (TR6 and TR7) for testing translation Look-aside buffer, which is memory cache used for speeding up the conversions of 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. The TR6 and TR7 control registers would be kept in

the subsequent i486 CPUs. However, by the time i586 ("Pentium") was released, the TR6 and TR7 MSRs were removed. As a result, software that was dependent on these control registers would no longer be able to execute on Intel Pentium series CPUs. At first there were only about a dozen of these MSRs (Model-Specific Registers), but lately their number is well over 200. Some MSRs have evidently proven to be sufficiently satisfactory and worth having due to their proven usability for debugging, program execution tracing, computer performance monitoring, and toggling of certain CPU features [30]. As a result, the Intel manual states that many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, are now deemed as permanent fixtures of the defined i386 architecture. For historical reasons (beginning with the Pentium 4 processor), these "architectural MSRs" were given the prefix "IA32_". One such MSR is the IA32 Extended Feature Enable Register (EFER). The proven usefulness of the EFER MSR has made Intel classify this MSR as architectural model-specific registers and has committed to their inclusion in future product lines.

Each MSR is a 64-bit wide data structure and can be uniquely identified by a 32-bit integer. For example, the IA32_EFER MSR can be uniquely identified by the 32-bit integer 0xC0000080. It is possible for a subset of the 64-bit wide MSR data structure to be reserved, so that it cannot be modified by a user. Non-reserved bits can be set or unset by using Intel's provided WRMSR instruction. Finally, any bit (reserved and 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 is able to enable and disable the syscall instruction. This bit is also interesting because it can be modified by any user who has privileges to execute the WRMSR instruction. For example, if the bit 0 is set to a value of 0, then the SYSCALL instruction will be undefined by the CPU. Therefore, every attempt to execute the SYSCALL instruction by a machine will result in an invalid OP CODE (#UD) exception (as previously discussed in the exception subsection). If bit 0 is set to 1, then the SYSCALL instruction will be defined by the CPU. By default

(unless manipulated by a user), bit 0 of the IA32_EFER MSR has bit 0 set to 1. That is why system calls are able to execute perfectly fine on our machines.

For our understanding our thesis, it is very important that we mention that according to Chapter 35 of Volume 3 of the Intel Architectures SW Developer’s Guide, MSRs hold three properties. Firstly, MSRs with a scope of ”thread” are separate for each logical processor and can only be accessed by the specific logical processor.

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. MSRs with a scope of ”package” are global to the package, so access from any core or thread context in that package will access the same register.

Table 2.1: 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

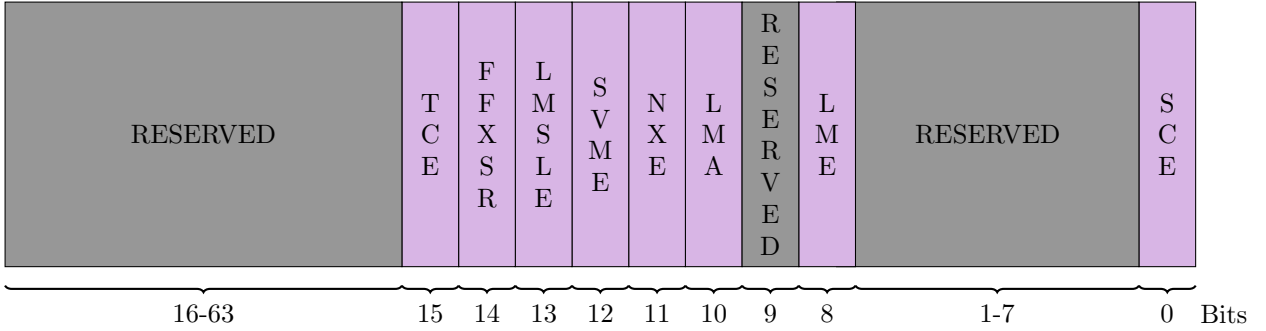


Figure 2.5: Representation of the IA32_EFER MSR (0xC0000080)

2.3 Intel Virtualization Extension (VT-X)

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

2.3.1 Overview

Intel VMX can be viewed as a function that switches processing from a VM to the hypervisor upon detection of a sensitive instruction by the physical CPU [8]. If a guest VM is able to execute sensitive instructions on a guest system without any intervention by the host, it will cause serious problems for both the hypervisor and guest VM [8]. Therefore, it is necessary for the physical CPU to detect that the execution of a sensitive instruction is beginning and to direct the hypervisor to execute that instruction on behalf of the guest VM. However, x86 CPUs were not originally designed with the need for virtualization in mind, so there exist sensitive instructions that the CPU cannot detect when a guest VM executes them []. As a result, the hypervisor is unable to execute such instructions on behalf of the guest system. Intel VT-x was developed in response to this problem [8].

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

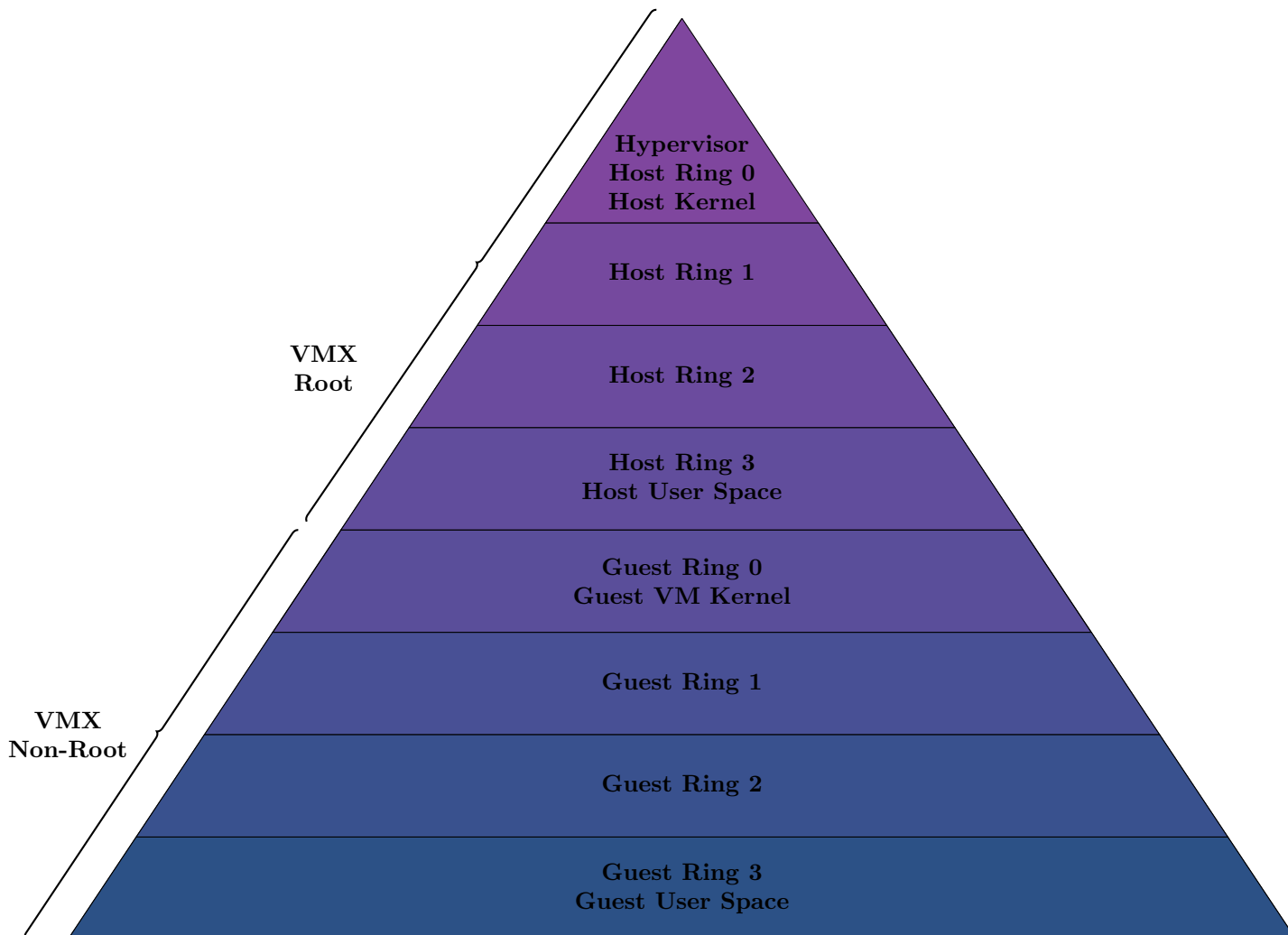


Figure 2.6: Illustration of VMX Root & Non-Root Mode in Relation to Intel Protection Rings.

2.3.2 Novel Instruction Set

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

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

VMXON

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

VMLAUNCH

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

VMPTRLD

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

VMPTRST

Stores the current VMCS pointer into a memory address

VMCLEAR

When a pointer to an active VMCS is given as operand, the VMCS becomes non-active. [4]

VMREAD

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

VMWRITE

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

VMCALL

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

VMRESUME

Enters non-root mode for an existing VM instance.

VMFUNC

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

VMXOFF

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

2.3.3 The Virtual Machine Control Structure (VMCS)

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

on the physical processor at a time.

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

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

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

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

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

2.3.4 VM-Exit

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

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



```

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

```

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

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

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

Table 2.2: Instructions that could cause conditional VM-exits as defined by the VM-exit control section of the VMCS

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

Table 2.2 – Continued From Previous Page

Instruction
MWAIT
PAUSE
RDMSR
WRMSR
RDPMC
RDRAND
RDSEED
RDTSR
RDTSRCP
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 2.3: Intel VMX Defined VM-Exits

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

Table 2.3 – Continued From Previous Page

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

Table 2.3 – Continued From Previous Page

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

Table 2.3 – Continued From Previous Page

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

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

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

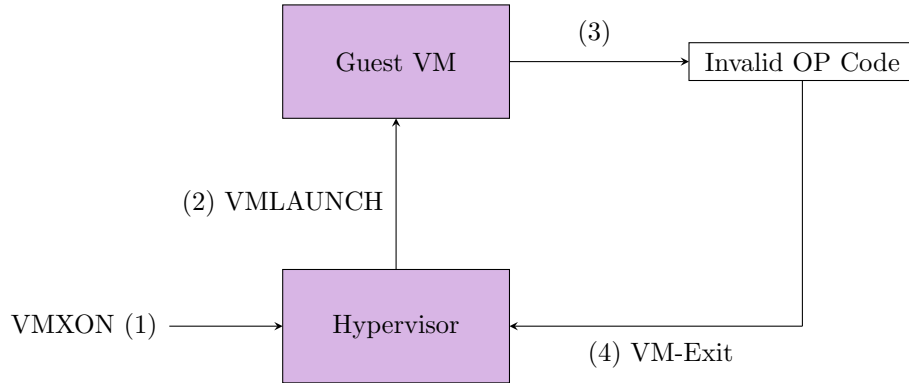


Figure 2.7: Life Cycle of a VM-Exit on invalid opcode

2.3.5 VM-Entry

VM-entry transfers control from the hypervisor (VMX root mode) back to the guest VM (VMX non-root mode). Software can enter VMX non-root operation using either of the VM-entry instructions VMLAUNCH and VMRESUME. For example, if the guest VCPU is not yet running (due to a prior VMCLEAR instruction), then it will use VMLAUNCH. In the case of a VM-exit, it will use VMRESUME [31]. Before a VM-entry can commence, the hypervisor executes dozens of checks to ensure that the state of the VMCS is correctly configured such that the subsequent VM-exit can be supported, and and the guest conforms to IA-32 and Intel 64 architectures [8].

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

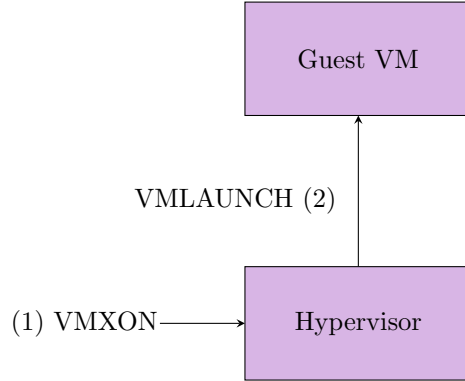


Figure 2.8: Life Cycle of a VM-Entry

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

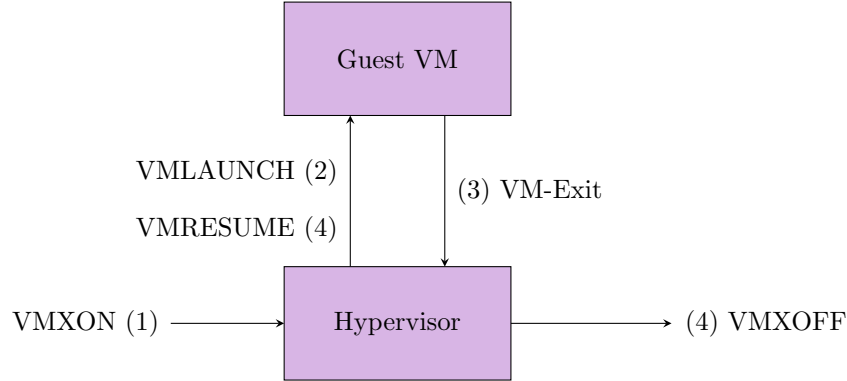


Figure 2.9: Successful Hypervisor Life Cycle Under Intel VMX

2.4 System Calls

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

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

2.5 The Kernel Virtual Machine (KVM) Hypervisor & QEMU

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

called `kvm.ko`, and is architecture independent [5]. The second KVM kernel module is architecture dependent [5]. Therefore, if the machine's physical CPU is Intel based, `kvm-intel.ko` will be inserted into the Linux kernel. If the machine's physical CPU is AMD based, then `kvm-amd.ko` will be inserted [5]. The insertion of the two kernel modules transforms the Linux kernel into a hypervisor. KVM was merged into the mainline open-source Linux kernel in version 2.6.20, which was released on February 5, 2007. Since its inception into the Linux kernel, Linux kernel developers have helped extend the functionality of KVM [8]. This section begins by explaining how KVM works and describes its internal and external components. KVM requires a CPU with hardware virtualization extensions, such as Intel VT-x or AMD-V. Our discussion will assume that KVM is utilizing Intel VMX virtualization extension.

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

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

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

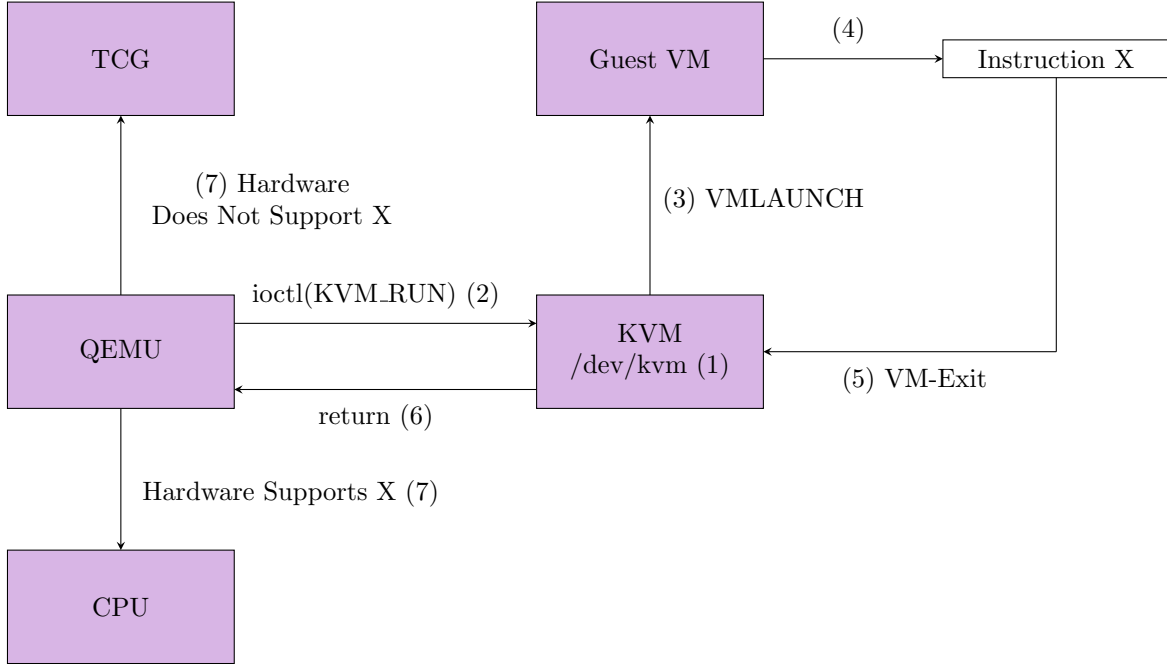


Figure 2.10: Decision on Whether QEMU use TCG or CPU for Executing an Arbitrary Instruction X.

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

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

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

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

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

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

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

    if (efer & EFER_LMA) {
```

```

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

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

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

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

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

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

```

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

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

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

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

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

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

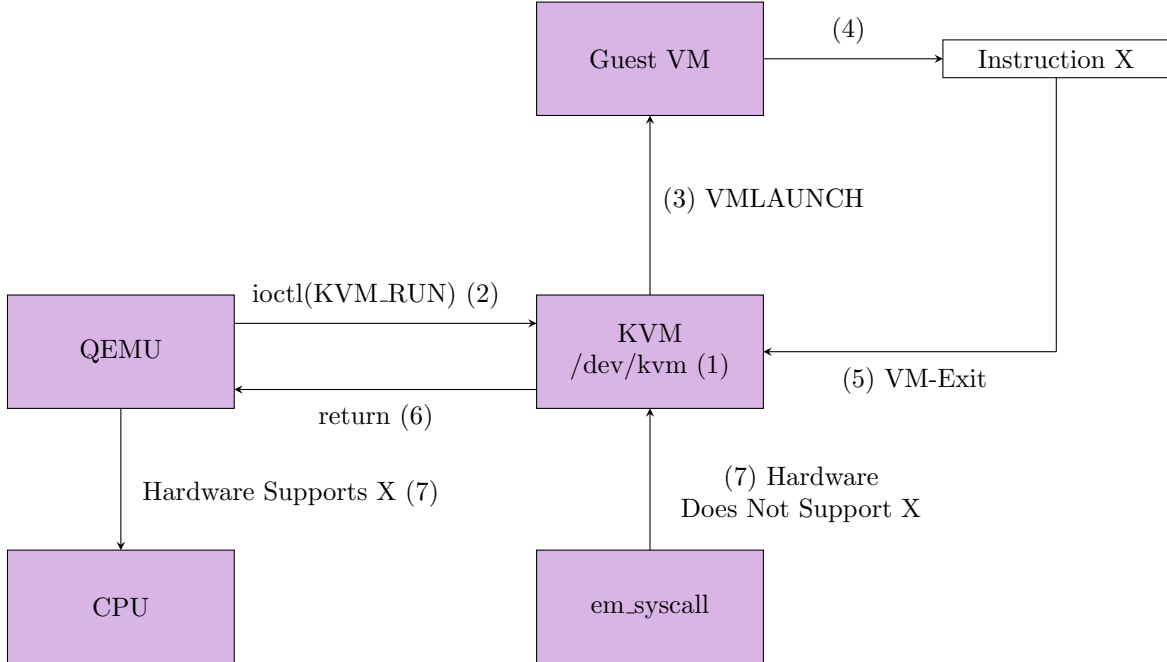


Figure 2.11: Partial KVM Life Cycle if TCG is Disabled

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

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

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

2.6 Virtual Machine Introspection

Virtual machine introspection (VMI) is a term created by Garfinkel and Rosenblum in 2003 [7]. VMI describes the method of monitoring and analyzing the state of a virtual machine from either the hypervisor level or the guest VM all without affecting its

functionality. However, due to the existence of hypervisors, VMI is now almost always implemented as an out-of-VM monitoring system [3]. Also, out-of-vm based monitors have been widely adopted because they run at higher privilege level and are isolated from the guest VMs that they monitor and can trap all the guest OS events as they are one layer below the guest OS. VMI allows us to take advantage of both the machine's hardware and the VMM to inspect any guest VM. The VMM is able to be manipulated ways that result important guest VM events are trapped to hypervisor. This ability to do this with a hypervisor is valuable for virtual machine introspection as it allows us to trap important actions a guest VM may execute, and inspect the guest's state at exactly that moment.

2.7 eBPF

2.7.1 Overview

eBPF programs are a more recent invention for accessing services and hardware from the Linux kernel space. Already these programs have been used for networking, debugging, tracing, firewalls, and more.

So what does eBPF do? It lets programmers safely execute custom bytecode within the Linux kernel without modifying or adding to kernel source code. While still a far cry from replacing LKMs as a whole, eBPF programs introduce custom code to interact with protected hardware resources with minimal threat to the kernel.

Born out of a need for better Linux tracing tools, eBPF drew inspiration from dtrace, a dynamic tracing tool available primarily for Solaris and BSD operating systems. Unlike dtrace, Linux could not get a global overview of running systems, it was limited to specific frameworks for system calls, library calls, and functions. Building on the Berkeley Packet Filter (BPF), a tool for writing packet-filtering code using an in-kernel VM, a small group of engineers began to extend the BPF backend to provide a similar set of features as dtrace. First released in limited capacity in 2014 with Linux 3.18, making full use of eBPF requires at least Linux 4.4 or above.

2.7.2 How Does eBPF Work?

From the perspective of a user, the eBPF workflow is surprisingly simple. Users can elect to write eBPF bytecode directly (not recommended) or use one of many front ends to write in higher level languages that are then used to generate the respective bytecode. IOVisor's bcc [29] offers bindings for several languages including Python, Go, and C++; users write eBPF programs in C and interact with bcc's API in order to generate eBPF bytecode and submit it to the kernel.

In Figure 2, we see a simplified visualization of how eBPF works. Before being loaded into the kernel, the eBPF program must pass a certain set of requirements. Verification involves executing the eBPF program within the virtual machine. Doing so allows the verifier, with 10,000+ lines of code, to perform a series of checks. The verifier will traverse the potential paths the eBPF program may take when executed in the kernel, making sure the program does indeed run to completion without any looping that would cause a kernel lockup. Other checks, from valid register state, program size, to out of bound jumps, must also be met. If all checks are passed, the eBPF program is loaded and compiled into the kernel at a point in a code path and listens for the right signal. That signal comes in the form of an event that passes where the program is loaded in the code path. Once triggered, the bytecode executes and collects information as per its instructions.

2.8 The Linux Kernel Tracepoint API

2.8.1 Overview

A Tracepoint is a marker (a piece of code) that can be hooked to certain areas of the Linux kernel source to allow for tracing kernel events at runtime and without stopping the execution of the kernel. Tracepoints are used by a number of tools for kernel debugging and performance problem diagnosis like eBPF. Although using tracepoints is ideal when possible, they have a few caveats; in particular, a limited number of tracepoints are defined by the kernel, and they do not cover an exhaustive list of kernel functionality. The official kernel code base consists of thousands of predefined events. A small

proper subset of these predefined events are KVM related. Whether that number will grow significantly is a matter of debate within the official team of Linux kernel developers community. However, as the Linux kernel is open-source, it is trivial to extend the tracepoint API to hook kernel functions in order to trace kernel events of interest.

2.8.2 Identifying 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 available for tracing. On Linux kernel version 5.18.8, there are 124 subsystems that are traceable by the API, which consist of the following:

2.8.3 Identifying Tracepoint Events

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

2.8.4 Tracepoint Format File

Each event has a format file that provides a user with arguments that can be traced from user space programs. For example, the format file for KVM exits can be found in `/sys/kernel/debug/tracing/events/kvm_exit/format`. These arguments come from default kernel space functions and are passed on to the kernel space tracepoint function. Information that these arguments contain is what is stored and sent back to the user space via buffers.

2.8.5 Using Tracepoint Events with eBPF

There are two segments in an eBPF program. The user space program takes care of the kernel space program declaration, attaching the tracepoint, and receiving and processing data sent through the buffer. The kernel space part of the program makes sure that the data passed from the tracepoint argument to the function is stored and transmitted back to the user space.

However when it comes to the USP, it is coded in python for programs that use the python API called bcc, and when using bpftrace, it is a bpf syntax that attaches the tracepoint/probe to the KSP.

eBPF gives us a way to connect to tracepoint events from the user space. BCC, which is a python API for eBPF gives us a trivial way to utilize the arguments given by the tracepoint events format file to trace an event.

2.9 pH-based Sequences of System Call

Designing Frail

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

3.1 Tracing KVM VM System Calls

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

Our design is based on trapping and emulating instructions. In other words, we

make every system call instruction in the KVM guest result in a VM-exit to the hypervisor. We do this by unsetting bit 0 of the IA32_EFER MSR using the WRMSR instruction. Recall in section 2.2.6, we discussed that the IA32_EFER MSR is capable of making the SYSCALL instruction undefined by the CPU if bit 0 is unset. According to the Intel 64 and IA-32 Architectures Software Developer’s Manual, when bit 0 of the IA32_EFER is unset, then every SYSCALL instruction will result in a #UD exception. Recall from section 2.2.2 that a #UD exception is a fault that traps execution to root mode. In the case of a VM, a #UD exception that occurs in VMX non-root, will result in a trap such that execution is transferred to VMX root, so that the #UD exception can be handled by the KVM hypervisor. Remember, the #UD exception occurred because the KVM VM SYSCALL, SYSRET, SYSENTER instructions were made undefined by the CPU. For that reason, we have two choices to handle the #UD exception. (1) We can utilize QEMU’s TCG capabilities to emulate the SYSCALL, SYSRET, SYSENTER instructions and VM-entry back into the KVM VM. (2) We can utilize KVM’s emulation capabilities (see Section 2.5) to emulate the instruction and then resume execution of the VM with a VM-entry. Our design chose to do the latter because emulating instructions via TCG is slower than emulating via KVM’s predefined emulation functions. 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. In other words, each VCPU of a KVM VM has its own IA32_EFER MSR. For that reason, to trace every KVM guest system call of a particular KVM VM, we unset bit 0 of the IA32_EFER MSR for every VCPU that exists on the KVM VM. We do this step for every KVM VM that exists. How do we know that a VM-exit was caused by a system call instruction, and not something else? In our design two checks are done to verify that a VM-exit was caused by a system call instruction. Recall that every #UD exception causes a VM-exit with code 0. Therefore, we filter out of every VM-exit code except for code 0. However, system call instructions are not the only instructions that result in a code 0 VM-exit. A code 0 VM-exit occurs when an NMI was delivered to the CPU. An NMI can be either a #UD exception, #BR exception, #BP exception, or #OF exception. Also, a #UD exception is not exclusively caused by an undefined system call. It can be caused by any undefined instruction to a CPU. Therefore, our second check consists of

checking the `%rip` instruction pointer. Recall that the instruction pointer `%rip` points to the next instruction (opcode) to execute. However, due to the semantic gap between the hypervisor and VMs, the `%rip` register corresponds to a VM virtual address. We need to get the physical address first. We do this by converting the guest address to its corresponding physical address, and then getting the value pointing to `%rip` using a KVM built-in function called `kvm_read_guest_virt()`. Now, we need to check if the first two bytes of the value that `%rip` points to is equal to `SYSCALL` (`0x0F05`), `SYSENTER` (`0x0F05`), or `SYSRET` (`0x0F07`). With these two checks, we can guarantee that the instructions that we trace are only x86 defined system call instructions. This approach allows a VMI system to introspect guest system call events in the ideal way: the guest VM can stay running throughout introspection.

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

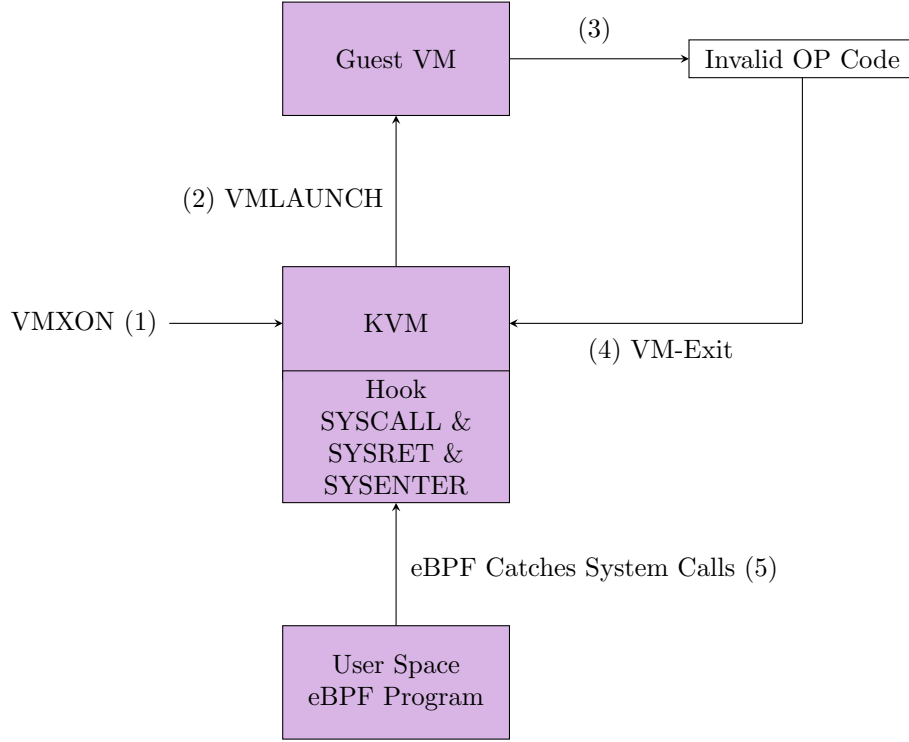


Figure 3.1: Illustration of Tracing KVM VM System Call

3.2 Tracing KVM VM Processes

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

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

a collision will never occur. The uniqueness of CR3s help with storing processes and their corresponding system calls. However, simply tracking changes of the CR3 register doesn't give us much insight into guest processes due to the semantic gap between the VM and hypervisor. In order to bridge this gap, we need to map every address that is loaded into the CR3 register to the name of the process (the binary). In order to get the process name, we solely track the exec family of system calls. Why do we do this? Because to create a new process, an exec type system call must be used. The first argument of every exec type system call requires the filename of the process. If we want to get the filename that was passed as an argument to exec, then we must read the %rdi register from the hypervisor level, which will store the virtual address of the 1st argument given to the exec function. We can then use KVM's builtin function `kvm_read_guest_virt` to read the virtual address given by %rdi to grant us the filename.

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

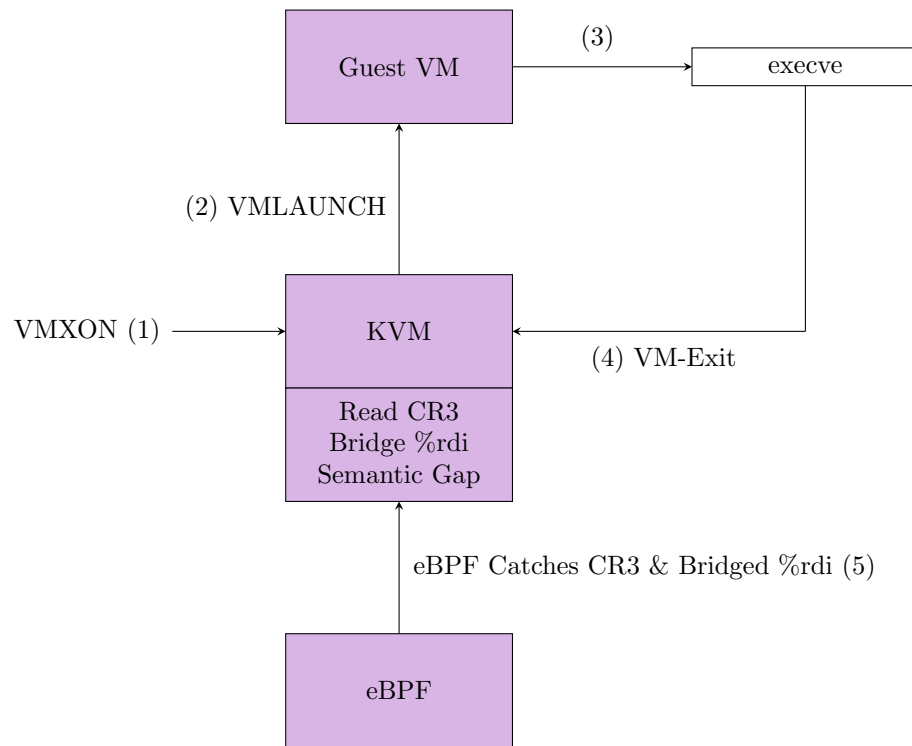


Figure 3.2: Illustration of Tracing KVM Guest Processes

Future Work (Winter 2022)

4.1 Implementing Frail

4.1.1 User Space Component

4.1.2 Kernel Space Component

4.1.2.1 Custom Linux Kernel Tracepoint

4.1.2.2 Kernel Module

4.1.3 Tracing Processess

4.1.4 Proof of Tracability of all KVM Guest System Calls

Bibliography

- [1] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. A survey on hypervisor-based monitoring: Approaches, applications, and evolutions. *ACM Comput. Surv.*, 48(1), aug 2015.
- [2] 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.
- [3] 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.
- [4] Ram Chandra Bhushan and Dharmendra K Yadav. Modelling and formally verifying intel vt-x: Hardware assistance for processors running virtualization platforms.
- [5] Humble Devassy Chirammal, Prasad Mukhedkar, and Anil Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.
- [6] 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.
- [7] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Ndss*, volume 3, pages 191–206. San Diego, CA, 2003.
- [8] Yasunori Goto. Kernel-based virtual machine technology. *Fujitsu Scientific and Technical Journal*, 47(3):362–368, 2011.
- [9] Charles David Graziano. A performance analysis of xen and kvm hypervisors for hosting the xen worlds project. 2011.

- [10] 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.
- [11] Shannon Meier, Bill Virun, Joshua Blumert, and M Tim Jones. Ibm systems virtualization: Servers, storage, and software. *IBM Redbook, May*, 2008.
- [12] 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.
- [13] Bryan D. Payne. *Virtual Machine Introspection*, pages 1360–1362. Springer US, Boston, MA, 2011.
- [14] 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.
- [15] 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.
- [16] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.
- [17] Anil Buntwal Somayaji. *Operating system stability and security through process homeostasis*. The University of New Mexico, 2002.
- [18] Paul C Van Oorschot. *Computer Security and the Internet: Tools and Jewels from Malware to Bitcoin*. Springer, 2021.
- [19] Jeffrey J. Wiley. *Protection Rings*, pages 988–990. Springer US, Boston, MA, 2011.
- [20] 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.