Guest-Based System Call Introspection with Extended Berkeley Packet Filter

by

Huzaifa Patel

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

Bachelor of Computer Science

Under the supervision of Dr. Anil Somayaji
Carleton University
Ottawa, Ontario
September, 2022

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

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

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 [16]. For example, healthcare companies are using the cloud to store patient records in databases [16]. Financial service companies are using the cloud for real-time fraud detection and prevention [16]. And finally, video game companies are using the cloud to deliver online video game services to millions of players around the world.

The existance of cloud computing can be attributed to virtualization. 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 [1]. The fundamental reason 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 [3]. A VM that is exposed to the Internet is changing constantly due to influx of non-determinisitc stream of data coming from the Internet and into the VM [2]. 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 [2]. The required capabilities and complexity of both computer programs and the system has led to a reduction in their reliability and security [2]. 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 [2]. As such, the role of a VM is highly security critical and its priority should be to maintain confidentially, integrity, authorization, availability, and accountability throughout its existance [13]. 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 [13]. 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 [4]. Frail is a VMI that (1) traces KVM guest system calls, (2) monitors malicious anomalies, and (3) responds to those malicious anomalies from the hypervisr 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 [2] 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 the corressponding 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 corressponding 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 confidentially, 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 systemss 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][2]. 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. The problem with Nitro is that it is now over 11 years old, and its official codebase has not been updated in over 6 years. For this reason, it is no longer compatible with any Linux 32-bit and 64-bit environments, and is not compatiable with newer Windows desktop versions. In fact, at the time of writing this 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 [3]. The primary reason for problem is that VMIs were designed in such a way that compromised compatibility and adaptability with subsequent versions of the OSs with which they were originally intended, implemented, and proven to be compatible with.

To solve the problem of incapability issues with VMI's like Nitro, we seek to design a spiritual successsor to Nitro that is intended to provide a VMI without sacrificing compatibility with subsequent versions of the Linux kernel. 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 [8].

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 privlige level than guests [7]. 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 corressponding 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 [7].

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 advantagous because the KVM hypervisor has less than one twelfth the number of lines of code than the Linux kernel; this smaller attack surface leads to fewer vulnerabilities in hypervisor-based VMI'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 [8]. 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) [7]. 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 [7].

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 [8].

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 [9]. As a result, malicious software (malware), such as kernel rootkits can be used to conduct privilege escalatation. 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 privlige, 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 privlige in the guest, the following scenario 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 processess 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 escaleted privlige 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 simplier, 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 [14]. 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 oppertunity 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 modifing its source code. However, this can lead to a plethora of security concerns, as running custom code in the kernel is dangerous and error prone. For example, if you make a logical or syntaxtical error in a user space application, it could crash the corressponding user space process. Likewise, if there exists a logical or syntaxtical error in kernel space code, the entire system could crash. Finally, if you make an error in an open source hypervisor code like KVM, all the running guest VM's could crash. The purpose of a VMI is to debug or conduct forensic analysis on a VM [15]. 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 corressponding 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. modifiing 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 datacenters 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 decistion 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 hardware 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 respeonding 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 traing 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 achitectures, 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 resistent. 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 archetecture, 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 crate, 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. [18]. 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.

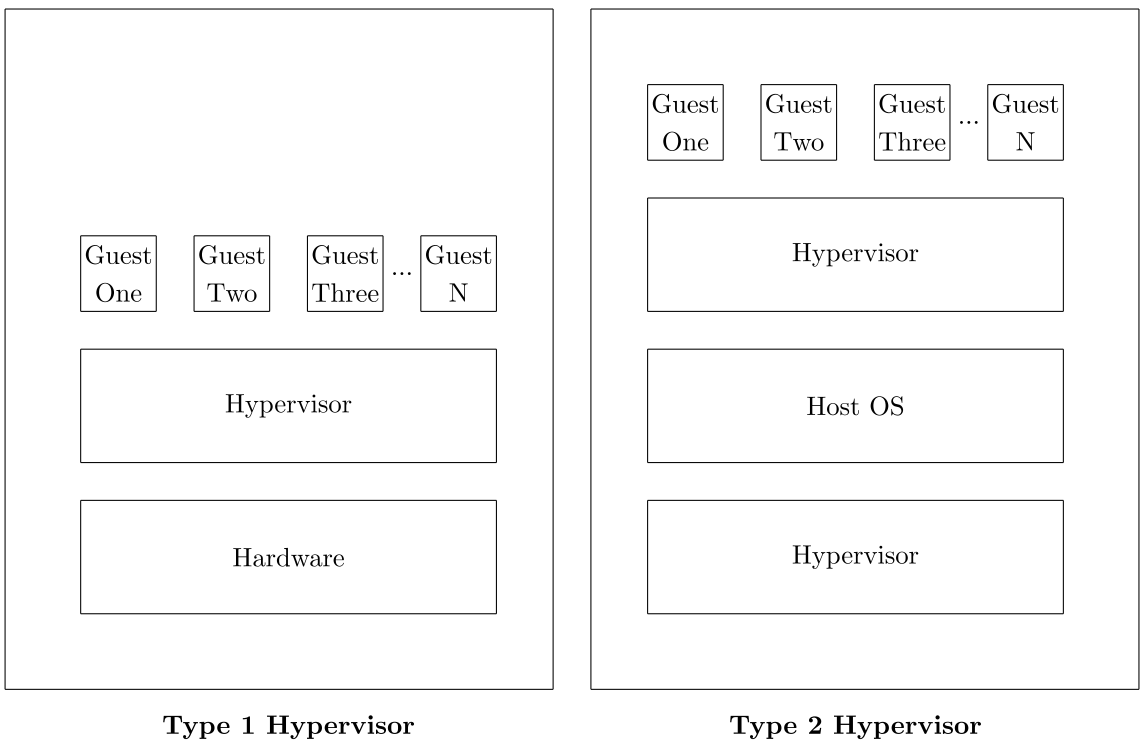

Figure 2.1: Mental Model of Type 1 & Type 2 Hypervisor

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 [19]. 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 [9]. With the new definitions, hypervisors can be classified into two types: (1) native hypervisors and (2) emulation hypervisors [9].

2.1.4 Native Hypervisor

Native hypervisors are hypervisos 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 [9]. 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 [9]. 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 [9].

2.2 Intel Central Processing Unit

2.2.1 Protection Rings

Before we explore the hypervisor further, we must introduce protection rings (also known as privlige modes, but not to be confused with CPU modes), which is a mechanism that Intel CPUs implement to aid in fault protection. 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 [22]. Prior to the implementation of protection rings, all system processes elements executed in the same processing space. This arrangement meant that when any process generated a fault, it had the ability to affect other processes that were running normally. This resulted in the process that caused the fault, as well as processes that did not generate a fault to crash [23]. Due to these problems, protections rings was introduced to provide the OS with a hierarchical layer for protecting the integrity and availability of both user space and kernel space processes. With protection rings, an OSs kernel can deal with faults by terminating only the process that caused the fault.

By creating a conceptual model for protection rings, one can better understand them. Therefore, we descibe protection rings as a hierarchical system that consists of four layers: Ring 0, Ring 1, Ring 2, and Ring 3. Next, we describe how portions of the OS are separated into each of the 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 comminicate 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 [23].

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 differenciate 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 resrouces from more privliged 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.

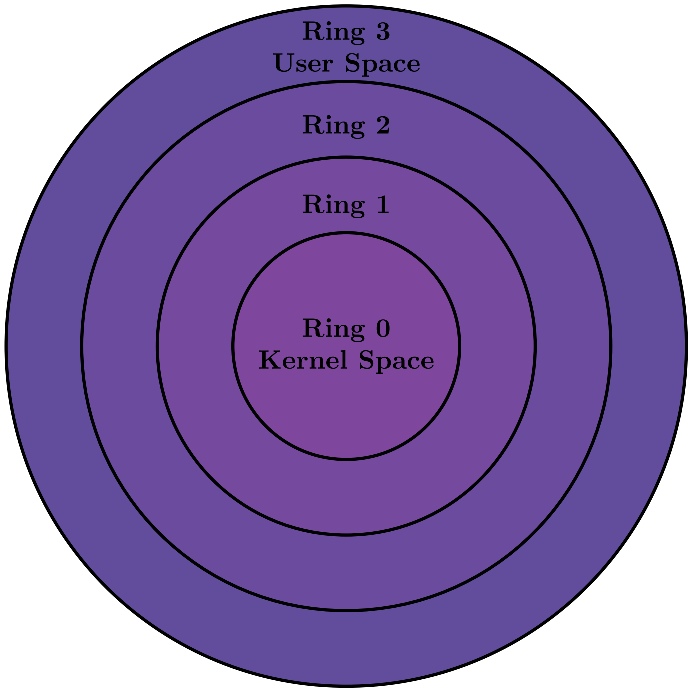

Figure 2.2: Illustration of the Intel x86 Protection Ring

2.2.2 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 occured during the execution of a program.

Exceptions can be divided into three categories:

- Faults
- Traps
- Aborts

However, we will only introduce faults and traps that are relevant to our VMI system's design and implementation. Aborts are not relevant to our VMI.

2.2.2.1 Faults

As previously mentioned, IEEE formally defines a fault as an error in a computer program's step, process, or data [22]. There exists many different types of faults, which are each initiated for different reasons. However, we will only introduce the Invalid Opcode (#UD) exception due to its relevance to the design and implementation of our VMI system.

2.2.2.1.1 Invalid Opcode

An illegal opcode, also called an undefined instruction is a fault that is generated due to an instruction to a CPU that is not supported by the CPU. The effect of executing an instruction that is undefined by the CPU results in a trap to an illegal opcode error handler. A CPU instruction that is mentioned in official documentation released by the CPU's designer or manufacturer can result in an illegal opcode if a user manipulates a Model Specific Register (MSR) so that a specific CPU instruction becomes undefined.

2.2.2.2 Traps

A trap is a synchronous interrupt triggered by a user process. A trap changes the mode of an OS from user to kernel mode. During a trap, the execution of a process is set as a high priority compared to user code. When the OS detects a trap, it pauses the user process, and executes the relevant trap handler inside the kernel. There exists many different types of traps, which are each initiated for different reasons. However, we will only introduce the single stepping trap due to its relevance to the design and implementation of our VMI system.

2.2.2.2.1 Single Stepping

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, it will do this for every instruction. Any program can activate single stepping by using a debugger such as GNU Debugger (GDB). When single stepping is enabled, there is no need to put a breakpoint to a specific line of code because every instruction will cause a trap. Instead, you can rely on the CPU to do the execution implicitly.

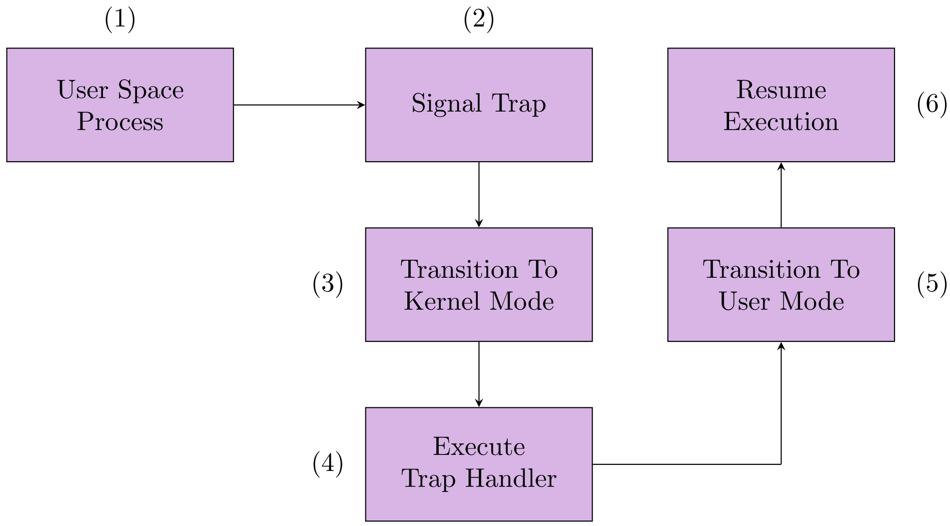

Figure 2.3: Trapping Life Cycle

2.2.3 CPU 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.

2.2.4 Model Specific Register (MSR)

A Model specific register (not to 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 usabililty 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 IA32_EFER MSR allows enabling or disabling the SYSCALL/SYSRET instruction, and also for entering and exiting long mode. 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 uniqely identified by a 32-bit integer. For example, the IA32_EFER MSR can be uniqely identified by the number 0xC0000080. It is possible for a subset of the 64-bit wide 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. Any bit (reserved and non-reserved) of the 64-bit wide data structure 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 integer that uniqely identifies an MSR.

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

Continued on next page

Table 2.1 – Continued From Previous Page

Bits(s)	Label	Description
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

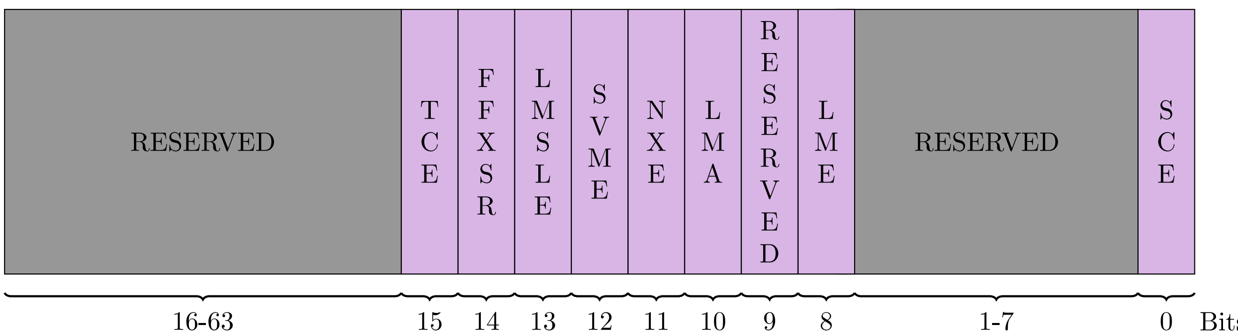

Figure 2.4: Representation of IA32_EFER MSR (0xC0000080)

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 "VMX".

2.3.1 Overview

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 anagalous 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-existant. 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 applicaitons 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.

2.3.2 Novel Intrusction 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 iteracting and manipulating the VMCS from root mode (hypervisor level). These include the VMXON, VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMLAUNCH, VMRESUME, and

VMXOFF instructions. Secondly, another subset of the new instructions were created for use by the the guest VM (non-root mode). These include the VMCALL, and VM-FUNC instructions. Lastly, there are 2 instructions that are used for manipulating translation lookaside buffer. These include the INVEPT and INVVPID instructions. Translation lookaside buffer is not relevant to this thesis. Therefore, we will not explain the INVEPT and INVVPID instructions.

VMXON

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

VMLAUNCH

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

VMPTRLD

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

VMPTRST

Stores the current VMCS pointer into a memory address

VMCLEAR

When a pointer to an active VMCS is given as operand, the VMCS becomes non-active. [26]

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 transfer control from the Guest VM (non-root mode) back to the hypervisor (root mode) and populate certain fields in the VMCS to inform the hypervisor about what triggered the VM-exit. 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). VM-exits can also occur conditionally (e.g., based on control bits in the VMCS). For example, the hypervisor can set a bit in a specfic field of the control section of the VMCS such that whenever a VM guest VCPU encounters a system call, a VM-exit to the hypervisor is performed.

To illustrate the purpose of the VMCS in relation to VM-exits, we will briefly describe a subset of fields located in the VM-exit information section of the VMCS.

There are Five major steps of a successful VM-Exit:

1. Save the VCPU state to the Guest state section of the VMCS.
2. Store information about the reason for the VM-Exit into the VM-Exit Information section of the VMCS.
3. Trap to the hypervisor (VMEXIT).

4. The hypervisor will handle the instruction(s) that the guest OS could not execute.

5. Save the result for the guest.

6. Save its own CPU state and perform VM-Entry (VMRESUME).

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

Table 2.2: Intel VMX Defined VM-Exits

VM-Exit Code	**Corresponding Name**
0	Exception or NMI
1	External interrupt
2	Triple fault
3	INIT signal
4	Start-up IPI
5	I/O SMI
6	Other SMI
7	Interrupt window
8	NMI window
9	Task switch
10	CPUID
11	GETSEC
12	HLT
13	INVD
14	INVLPG
15	RDPMC
16	RDTSC
17	RSM
18	VMCALL
19	VMCLEAR

Continued on next page

Table 2.2 – Continued From Previous Page

VM-Exit Code	Corresponding Name
20	VMLAUNCH
21	VMPTRLD
22	VMPTRST
23	VMREAD
24	VMRESUME
25	VMWRITE
26	VMXOFF
27	VMXON
28	CR access
29	MOV DR
30	I/O Instruction
31	RDMSR
32	WRMSR
33	VM-entry failure 1
34	VM-entry failure 2
36	MWAIT
37	Monitor trap flag
39	MONITOR
40	PAUSE
41	VM-entry failure 3
43	TPR below threshold
44	APIC access
45	Virtualized EOI
46	GDTR or IDTR
47	LDTR or TR
48	EPT violation
49	EPT misconfig
50	INVEPT
51	RDTSCP
52	VMX timer expired

Continued on next page

Table 2.2 – Continued From Previous Page

VM-Exit Code	Corresponding Name
53	INVVPID
54	WBINVD/WBNOINVD
55	XSETBV
56	APIC write
57	RDRAND
58	INVPCID
59	VMFUNC
60	ENCLS
61	RDSEED
62	Page-mod. log full
63	XSAVES
64	XRSTORS
66	SPP-related event
67	UMWAIT
68	TPAUSE
69	LOADIWKEY

To synthesise all the information above, 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. 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 not being defined by the CPU designer, or because a user manipulated the CPU MSR to disable the CPU instruction that considered undefined. The effect of executing an instruction that is undefined by the CPU results in a trap to an illegal opcode error handler.

First, the physical CPU begins in root mode, and executes the VMXON instruction to start virtualisation. In Figure 2.4, the hypervisor starting virtualization is illustrated in (1). Next, the hypervisor executes a VMLAUNCH instruction in order to pass execution to the guest VM (non-root mode). In Figure 2.4, the hypervisor starting virtualization is illustrated in (2). The VM instance runs its own code as if running natively until it attempts something that is prohibited, that causes a VM-exit and a switch to the hypervisor (root mode). In our example, the guest ran an undefied instruction. The hypervisor will consult the VMCS to look into why the VM-exit caused. Based on the information provided by the VMCS, the hypervisor will take action to deal with the reason for the VM-exit. and then execute the VMRESUME instruction. In Figure 2.4, this is illutrated by (3).

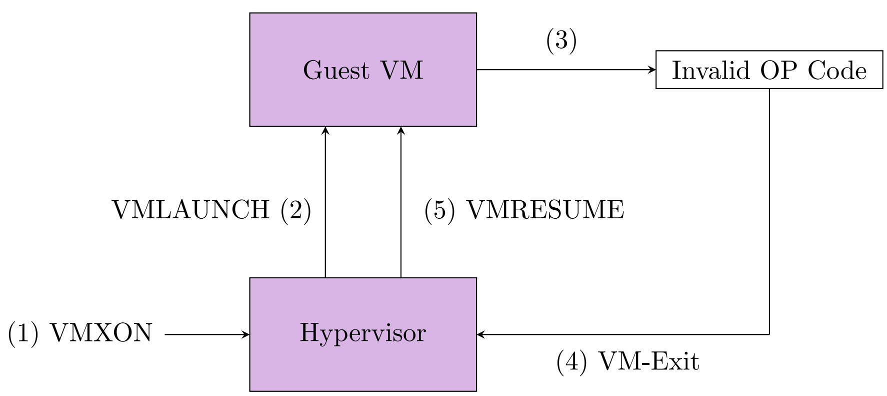

Figure 2.5: Life Cycle of VM-Exit on invalid opcode

2.3.5 VM-Entry

VM-entry transfers control from the hypervisor (root mode) back to the guest VM (non-root mode) and populate certain fields in the VMCS to inform the hypervisor about what triggered the VM-exit. 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). VM-exits can also occur conditionally (e.g., based on control bits in the VMCS). For example, the hypervisor can set a bit in a specfic field of the control section of the VMCS such that whenever a VM guest VCPU encounters a system call, a VM-exit to the hypervisor is performed.

2.3.6 Conclusion

Now that we have introduced the background information of Intel Virtualization, we can give an overview of the life cycle of the hypervisor executing on VMX. First, a program executing in root mode needs to execute the VMXON instruction to enable VMX.

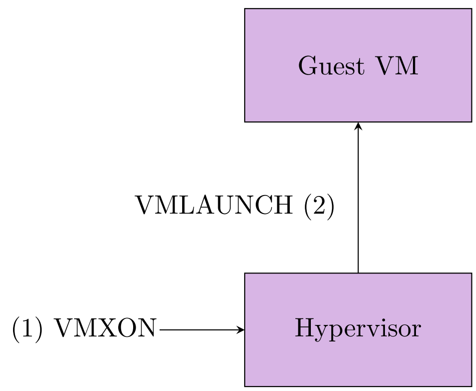

Figure 2.6: Life Cycle of VM-Entry

At this point, the program is considered a hypervisor. Second, the hypervisor sets up a valid VMCS with the appropriate control bits set. Third, the hypervisor can launch a VM with the VMLAUNCH instruction, transferring execution to the VM. If starting the VM did not fail, the hypervisor will now wait for the guest to trigger a VM-exit. Fourth, the hypervisor will regain control through a VM-exit triggered by the VM and react accordingly (unless launching failed, in which case VMLAUNCH would return an error). Fifth, the hypervisor transfers execution control back to the VM by executing the VMRESUME instruction, and we effectively go back to step (3). Alternatively, the hypervisor stops the VM and disables VMX by executing VMXOFF. This process is depicted in Figure 2.1.

2.4 Hypervisor Security Without Intel VT-X

Protection rings is a mechanism to protect data and functionality from faults (by improving fault tolerance) and malicious behaviour (by providing computer security). It is designed to have a hierarchical design that separates and limits the interaction between the user space and kernel space within and OS. It's purpose is to provide fault protection and tolerance among computer users, components, applications, and processes.

Each VM running on a hypervisor has its own security zone, which is not to be accessed by other existing VMs.

To explore the security issues related to Hypervisor security, first, we have to know about the various privilege modes that CPUs provide. The privlige modes are known as protection rings. There are three privilege level in any processor as shown in Figure x.x.

2.5 Hypervisor Security With Intel VT-X

Ring 3 is the least privileged, and is where normal user processes execute. In this ring 3, you cannot execute privileged instructions. Ring 0 is the most privileged ring that allows the execution of any instruction. In normal operation, the kernel runs in ring 0. Ring 1 and 2 are not used by any current operating system. However, hypervisors are free to use them as needed [6]. As shown in Fig. 1.1, the KVM hypervisor is kept in kernel mode (ring 0), the applications in user mode (ring 3), and the guest OS in a layer of intermediate privilege (ring 1). As a result, the kernel is privileged relative to the user processes and any attempt to access kernel memory from the guest Os program leads to an access violation. At the same time, the guest operating system's privileged instructions trap to the hypervisor. The hypervisor then performs the privileged instruction(s) on the guest OS' behalf.

2.6 The Kernel Virtual Machine Hypervisor

Kernel-based Virtual Machine (KVM) is a hypervisor that is implemented as a Linux kernel module that allows the kernel to function as a hypervisor. It was merged into the mainline Linux kernel in version 2.6.20, which was released on February 5, 2007. KVM requires a CPU with hardware virtualization extensions, such as Intel VT-x or AMD-V. While working with KVM, we will only be focusing on Intel VT-x hardware virtualization.

2.6.1 Model Specific Registers

2.6.2 VMCS

2.6.3 VM ENTRY Context Switch

2.6.4 VM EXIT Context Switch

2.7 QEMU

2.8 System Calls

2.9 Virtual Machine Introspection

VMI describes the method of monitoring and analyzing the state of a virtual machine from the hypervisor level [Nitro].

In general, a security monitoring system can be defined as M(S, P) $\rightarrow$ True,False, (1) where M denotes the security enforcing mechanism, S denotes the current system state, and P denotes the predefined policy. If the current state S satisfies the security policy P, then it is in a secure state (True), and if M is an online mechanism, it can allow continued execution. Otherwise, it is insecure (False); an attack1 is detected, and M can halt the execution (for prevention) or report that there is an attack instance. For example, in an antivirus system, S can denote the current memory and disk state, and P the signatures of viruses; if M identifies that there is any running process or suspicious file having one of the signatures defined in P, the antivirus will raise an alarm. In a system call–based intrusion detection system, S can denote the current system call and P can denote the correct state machines for S; if M identifies that S deviates from P, then it can raise an intrusion alert.

2.10 eBPF

2.11 The Linux Kernel Tracepoint API

2.12 pH-based Sequences of System Call

There are 12 projects that use the guest-assisted approach. The pioneer work, LARES [Payne et al. 2008], inserts hooks in a guest VM and protects its guest com- ponent by using the hypervisor for memory isolation with the goal of supporting active monitoring. Unlike passive monitoring, active monitoring requires the interposition of kernel events. As a result, it requires the monitoring code to be executed inside the guest OS, which is why it essentially leads to the solution of inserting certain hooks inside the guest VM. The hooks are used to trigger events that can notify the hypervisor or redirect execution to an external VM. More specifically, LARES design involves three components: a guest component, a secure VM, and a hypervisor. The hypervisor helps to protect the guest VM component by memory isolation and acts as the communica- tion component between the guest VM and the secure VM. The secure VM is used to analyze the events and take actions necessary to prevent attacks.

2.13 Nitro: Hardware-Based System Call Tracing for Virtual Machines

Designing Frail

Some VMI's introspect events like memory map and reads are done in a nonideal way: the events are introspected by a VMI system by halting the guest (pause-and-introspect) instead of accessing guest memory contents while the guest VM is running. This significantely hinders the overall performance the virtualization environment. Similarly, depending on the event, a VMI can only examine data trail during off-peak hours, so the guest VM can constantly stay active. With this type of VMI implementation, there is a chance, that a particularly successful intruder could tamper the audit trail and hide the intrusion before it is examined by the VMI. For this reason, a guest event that allows for computationally fast realtime introspection is useful.

If there exists an application programming interface (API) that maps a guest event to the hypervisor level, then a hypervisor-based VMI is capable of collecting an audit trail, and using that information to maintain the stability and security of a VM. For example, a VMI system can utilize a combination of guest process memory, guest processor instructions, a given guest user's keystrokes or commands, the guest systems resource usage, and of course guest system calls.

With system calls, the VMI can analyze the audit trail of an event, flag any unusual, anomalous, or prohibited behavior, and then initiate a response based on a security policy with a high success rate, and without hindering the overall performance of the virtualization environment. This can all be done live while the guest OS is still running, and is considered the most ideal case of introspecting a VM.

An evasion-resistant mechanism is a mechanism which is impossible for an attacker to circumvent when correctly implemented and deployed in an ideal system. Nitro defines a correctly implemented mechanism as a mechanism that perfectly enforces the policy that it was designed to enforce with no flaws or errors. In the same manner, we define an ideal system as a system that perfectly implements its design and contains no flaws or errors.

3.1 The Problem with Hypervisor based VMI's

The problems we face are strongly related to the six research questions we previously proposed.

3.1.1 The Semantic Gap Problem

The primary advantage of in-VM systems is their direct access to all kinds of OS level abstractions like files, and processes.

However, when using a hypervisor-based VMI system, access to all of the rich semantic abstractions that the OS provides is lost. Although hypervisors have a grand view of the entire state of the VMs they monitor, this grand view unfortunately is provided with hardware-level abstractions, which consists ones and zeros, putting a disadvantage to a humans due to providing no context. The disparity between OS and hardware level abstractions is known as the semantic gap. As we are using a hypervisor-based VMI, guest system call and process information can only be detected based on register values.

As an example of how the semantic gap creates challenges for introspection, consider how a hypervisor might go about listing the processes running in a guest OS. The hypervisor can access only hardware-level abstractions, such as the CPU registers and contents of guest memory pages. The hypervisor must identify specific regions of guest OS memory that include process descriptors, and interpret the raw bytes to reconstruct semantic information, such as the command line, user id, and scheduling priorities.

3.1.2 Inability to Trace KVM Guest System Calls from the KVM Hypervisor

One of the problems with hypervisor-based VMI systems is that not all the guest events result in the guest trapping to the hypervisor. For instance, guest system calls do not result in the guest trapping to the hypervisor. For this reason, by default, it is not possible to trace system call KVM VMs from the hypervisor. For this reason, it is not feasible for eBPF to observe guest system calls.

3.2 Approaching the Problem

3.2.1 Approaching The Semantic Gap Problem

3.2.2 Approaching the KVM Hypervisors inability to Trace Guest System Calls

To observe system calls from the guest operating system, we must force system call instructions to result in a VM Exit. To achieve this, we must unset the system call enable (SCE) bit of the guest VMs Extended Feature Enable Register (EFER), which is a Model Specific Register (MSR). Unsetting this bit results in system call instructions being unknown to the CPU. As a consequence, when system call instructions are executed in guest VMs, an invalid opcode exception (#UD) is generated that induces a VM Exit with exit reason zero. From this point, eBPF can be used to observe VM Exits from the host, and the RIP register can be used to verify that the VM Exit with reason 0 was due to a system call instruction. As unsetting the SCE bit results in system call instructions to be unknown by the CPU, we will need to explictly emulate every system call instruction in the hypervisor before making an entry back into the VM.

Implementing Frail

4.1 User Space Component

4.2 Kernel Space Component

4.2.1 Custom Linux Kernel Tracepoint

4.2.2 Kernel Module

4.3 Tracing Processess

4.4 Proof of Tracability of all KVM Guest System Calls

Threat Model of Frail

Future Work

6.1 Disadvantage to our Design

Model specific registers are used to provide access to features that are generally tied to implementation dependent aspects of a particular processor.The features provided by the model specific registers are expected to change from processor generation to processor generation and may even change from model to model within the same generation. Because these features are implementation dependent, they are not recommended for use in portable software. Specifically, software developers should not expect that the features implemented within the MSRs will be supported in an upward or downward compatible manner across generations or even across different models within the same generation.

Conclusion

References

https://dl.acm.org/doi/pdf/10.1145/361011.361073 [1] https://people.scs.carleton.ca/ soma/pubs/soma-diss.pdf [2] https://dl.acm.org/doi/pdf/10.1145/2659651.2659710 [3] https://doi.org/10.1007/978-1-4419-5906-5_647 [4] https://doi.org/10.1007/978-3-642-25141-2_7 [5] modern operating systems andrew s. tanenbaum [6] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7299 [7] https://dl.acm.org/doi/pdf/10.1145/1655148.1655150 [8] https://dl.acm.org/doi/10.1145/2775111 [9] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7299979 [10] https://people.redhat.con KVM-monolithic.pdf [11] https://github.com/willfindlay/honors-thesis/blob/master/thesis/thesis.pdf [12] https://people.scs.carleton.ca/ paulv/toolsjewels/TJrev1/ch1-rev1.pdf [13] https://www.sciencedire [14] https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_647 [15] https://doi.org/10.1016/j.autcon.2020.103441 [16] https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide [17] https://www.redbooks.ibm.com/redpapers/pdfs/redp43 [18] https://doi.org/10.31274/etd-180810-2322 [19] https://www.techtarget.com/searchitoperations/defi machine-VM [20] https://stackoverflow.com/questions/39019501/understanding-kvm-cpu-scheduler-algorithm [21] https://link.springer.com/article/10.1007/s11334-017-0300-7 [22] https://link.springer.com/referenceworkentry/10.1007/978-1-4419-5906-5_788 [23] https://archive.wikiwix.com/cache/index2.php?rev_t=20101027065321&url=http%3A%2Fark.intel.com [24] https://wiki.osdev.org/VMX [25] https://www.researchgate.net/profile/Dharmendra-Yadav-12/publication/352877971_D6021048419/links/60dd847e299bf1ea9ed36e7f/D6021048419.pdf [26] https://www.felixcloutier.com/x86/vmread [27] https://www.felixcloutier.com/x86/vmwrite [28] https://www.felixcloutier.com/x86/vmcall [29] http://datasheets.chipdb.org/Intel/x86/Pentium/Er [30]