# 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 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-V** | Advanced Micro Devices Virtualization |
| **VT-x** | Intel Virtualization Extension |
| **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 |

# 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. For example, healthcare companies are using the cloud to store patient records in databases. Financial service companies are using the cloud to power real-time fraud detection and prevention. And finally, video game companies are using the cloud to deliver online 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: (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 1960s, it wasn't widely adopted until the early 2000s due to hardware limitations. 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 virtual machine isolated from the Internet may behave consistently over time. However, a VM exposed to the Internet is a new system every day, as the rest of the Internet changes around it [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. 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 previous action 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) framework that enhances the capabilities of related VMIs like Nitro. 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 responsible for the observed malicious anomaly. The tracing, monitoring, and responding of data 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.

## 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 then 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 KVM 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 hundreds, if not thousands 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, VMs should be programmed to watch over themselves through the hypervisor level. 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.

### 1.4.1 Why Design a New VMI?

The problem of protecting VMs dates back to 2003, when Garfinkel and Rosenblum first introduced the concept of VMI as a hypervisor-level Intrusion Detection System (IDS), which combined the advantages of both network-based and host-based IDS [10][2]. Since

then, widespread research and deployment of VMs has led to an abundent creation of VMI's, some more practical than others, for the purpose of limiting the damage that untrusted software can do to VMs. These are covered in more depth in the "Related Work" section. For now, we focus on why it might be necessary to design and implement yet another KVM VMI framework even though many of them already exist.

At the time of writing this thesis, to our knowledge, there is one relevant KVM VMI to our thesis named Nitro, 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 at the time of this writing, it is over 11 years old, and has not been updated in over 6 years. For this reason, it is no longer compatiable with any Linux 32-bit, and 64-bit environments, and is not compatiable with newer Windows desktop versions. As of writing this, Nitro only supports Windows XP x64 and Windows 7 x64, making it completely useless as Windows XP is a discontinued OS that is now over 21 years old and consists of only 0.39% of worldwide Windows desktop version market share. Similarly, Windows 7 is 13 years old, and consists of only 9.6% of worldwide Windows desktop version market share. There is a fundamental problem with the state of many existing VMI's like Nitro: When the codebase OSs change or kernels change, they are unable to solve the problem for which they were originally designed to solve: to trace and monitor VMs to protect the VM from being compromised in the event of a successful attack [3]. One of the primary reasons for this is that current VMIs were designed in such a way that compromised compatibility with subsequent versions of the OSs with which they were originally compatible with. To solve the problem of Nitro's incapabilities, 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 discuss this further in the "Contributions" section and "Background" chapter.

### 1.4.2 Why Design a Hypervisor-Based Virtual Machine Introspection?

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 implement-

ing a hypervisor-based VMI by analyzing the advantages and disadvantages of both hypervisor-based and guest-based VMI's.

**Hypervisor-Based VMI's**

Hypervisor-based monitors offer four key advantages over traditional guest-based VMI's: (1) isolation, (2) inspection, (3) interposition, and (4) deployability [8].

Isolation in our context, refers to the property that VMI's are less susceptible to tampering [8]. Hypervisor-based VMI's run at a higher privlige level than guest OS'. For this reason, they are isolated from the guest OS, which makes them tamper-resistant from the guest assuming that the hypervisor does not have its own vulnerabilities [7]. By isloating a VMI from a guest, it allows it to be immune from attacks that originate in the guest, even if the VMI is actively monitoring the attacked guest [7]. Due to the isolation of host-based VMI's from the guest, they only need to trust the underlying hypervisor instead of the entire Linux kernel. This 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 VMI's. Although attackers may still be able to generate false data by tampering the guest, the hypervisor-based VMI can be implemented to successfully defend against false data generation attacks .

The second key advantage, which is 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 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.

The third key advantage is interposition, which is the the ability to inject operations

into a running VM based on certain conditions. Due to the close relationship between a hypervisor and a host-based VMI, the VMI is capable of modifying any of the states of the guest and interfering with every activity of the guest. In terms of our VMI, interposition makes it easy to respond to the observed malicious anomalies by slowing down the process responsible for the malicious anomaly [7].

Deployability of a VMI refers to the ease with which it can be taken from development to production onto a system. It can be measured in terms of the number of discrete steps required to deploy VMI software 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 an account for any guest, and we do not need to install the VMI software, or any of its dependencies inside the guest. Instead, we only need to install dependencies of the VMI on the host. Afterwards, we may execute our VMI on the host during its runtime without disrupting any services in the host or guest.

**Guest-Based VMI's**

Although guest-based systems have been very successful with systems like [insert guest-based vmi here], they are more susceptible to three types of threats: (1) privlidge escalation, (2) spoofing, and (3) tampering [8].

Guest-based VMI's are not isolated as well as hypervisor-based VMI's, because they are executed on the same privlidge level as the VM(s) that they are protecting [9]. As a result, malicious software (malware), such as kernel rootkits that escalate privlidge can allow an attacker with unauthorized access to a VMI, or software that a VMI depends on. If successful, an attacker can tamper with all the components of a VMI. For example, an attacker can tamper with the tracing software that collects system call information and/or the corresponding process that requested the system call. Since our VMI depends on hooking specific kernel functions, attackers can modify the relevant symbols within the symbol table to bypass VMI properties. They can also tamper with the software that handles sequences of system calls, which is the monitoring tool that enforces our VMI's security policy. Finally, attackers can tamper with the software

that the VMI depends on like logs, and insert them with false data. Also, if the VMI is deployed using a kernel module in kernel space or a normal process in user space, attackers can simply remove or shut down the monitoring module or process. 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 many two fundamental advantages: (1) rich abstractions, and (2) fast speed.

There are plenty of abstractions for guest-based monitors from which to extract the OS and process state. They can use critical kernel variables and functions, system call and the guest process responsible for requesting the system call. At an even higher level of abstraction, they can also use the available guest-based security tools to extract system calls and processes. Unlike hypervisor-based VMI's, with guest-based VMI's, we are able to trivially intercept system calls, and processess, and inspect their sequences.

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 costly because it requires a VM-Exit, which we will discuss in subsequent chapters. The most effective way optimize the guest mode is to reduce the number of VM-Exits [11]. However, we believe that the disadvantages of guest-based VMI's outweigh its advantages. More specifically, the security of a VMI and the VM's that require protection by a VMI 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 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 detecting anomalies, and maintaining the fundamental goals of computer security. The demand for collecting particular Linux system data has led to the creation of many Linux 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 will crash the corressponding user space process. Likewise, if there exists any type of error(s) in the kernel code, the entire system will crash. Finally, with context to our research, if you make an error in open source hypervisor code like KVM, all the running guest VM's will 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 is not very beneficial. 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 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 termi-

nation, 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, and/or inserting custom kernel modules).

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 both saves memory and reduces the amount of time it takes to insert an eBPF program into the kernel. Additionally, since eBPF runs in the kernel space and communicates with user space via both predefined and custom Linux kernel tracepoints, 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 itself 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 Evastion 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, 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.

## 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

## 2.1 Virtual Machine Introspection

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

## 2.2 Hypervisor

There are two ways a hypervisor can virtualize a machine:

A hypervisor runs Guest OS instructions either directly on the host's CPU, or on the host OS. In both scenarios, the goal of a hypervisor is to provide a software-controlled layer that resembles the host hardware. Hypervisors can be classified into two types that are dependent on how they to runs Guest OS instructions.

(1) Type 1 (bare metal) hypervisors, which runs Guest OS instructions directly on the host's hardware in order to control the hardware and monitor the guest OS. Typical examples of such hypervisors include Xen, VMware ESX, and Microsoft Hyper-V.

(2) Type 2 (hosted) hypervisors, which run within a traditional OS. In other words, a hosted hypervisor adds a distinct software layer atop the host OS, and the guest OS becomes a third software layer above the hardware. Well-known examples of type 2 hypervisors include KVM, VMware Workstation, VirtualBox, and QEMU.

Although the preceding type 1 and type 2 hypervisor classification has been widely accepted, it is not clear it insufficiently differentiates among hypervisors of the same type (e.g., KVM vs. QEMU).

KVM is not a clear case as it could be categorized as either one. The KVM kernel module turns Linux kernel into a type 1 bare-metal hypervisor, while the overall system could be categorized to type 2 because the host OS is still fully functional and the other VM's are standard Linux processes from its perspective.

There-fore, based on how the virtualization gets designed (hardware vs. software) and the guest OS and its application code is executed, we can have another type of classification of hypervisors that will be used throughout this thesis:

(1) Native hypervisors that directly push the guest code to execute natively on the hardware using hardware virtualization.

(2) Emulation hypervisors that translate each guest instruction for an emulated execution using software virtualization.

Examples of native hypervisors include Xen, KVM, VMware ESX, and Microsoft HyperV, and emulation hypervisors include QEMU, Bochs, and the very early versions of VMware-Workstation and VirtualBox (note that recent VMware-Workstation and VirtualBox are able to execute the guest OS code natively). Since there is no binary code translation involved, native hypervisor runs much faster than emulation hypervisor.

In this thesis, we will be solely on the KVM VM.

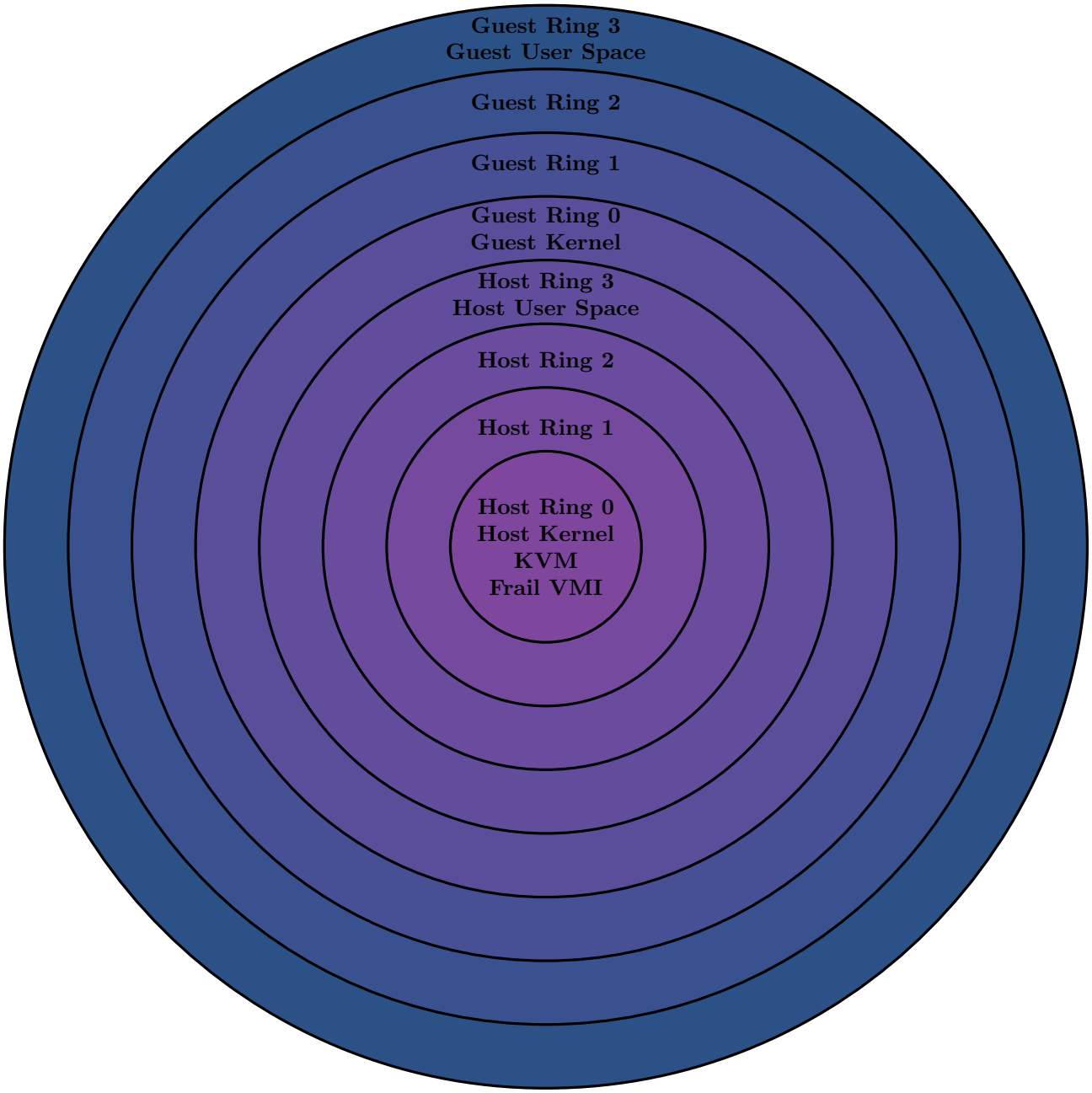
Hardware-assisted

## 2.3 Intel Virtualization Extention (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.

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

Figure 2.1: Illustration of the x86 Protection Ring of a VM




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, hyper-

visors 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.

## 2.4 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.

### 2.4.1 Model Specific Registers

### 2.4.2 VMCS

### 2.4.3 VM ENTRY Context Switch

### 2.4.4 VM EXIT Context Switch

## 2.5 QEMU

## 2.6 System Calls

## 2.7 Virtual Machine Introspection

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.

## 2.8 eBPF

## 2.9 The Linux Kernel Tracepoint API

## 2.10 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.

## 2.11 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

# 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 operat-
ing 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.com
KVM-monolithic.pdf [11] https://github.com/willfindlay/honors-thesis/blob/master/thesis/thesis.pdf
[12]