NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

HIGH-FIDELITY VIRTUALIZATION FOR CYBER OPERATIONS

by

Corey D. Ingraham

June 2019

Thesis Advisor: Alan B. Shaffer
Co-Advisor: Gurminder Singh

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2019	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE
HIGH-FIDELITY VIRTUALIZATION FOR CYBER OPERATIONS

5. FUNDING NUMBERS

6. AUTHOR(S) Corey D. Ingraham

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Secretary of Defense, WASHINGTON, DC 20301; Office of the

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Virtualization has become ubiquitous in cyberspace over the last decade, expanding into cloud technology and embedded mobile devices. Outside of a highly sophisticated user community, consumer demand has yet to realize the need for creation of a high-fidelity virtual environment, resulting in a lack of specialized hypervisor design and a stall in the evolution of a higher level of fidelity in virtual hardware.

Most modern hypervisors rely on virtual hardware designed to meet the minimum requirements of computing, leading to a low-fidelity implementation in virtual systems when compared to their non-virtualized counterparts. High-fidelity can be achieved by effectively emulating these physical characteristics in a virtual environment that could enable further development of cyber operations by providing all the benefits of virtualization coupled with a specialized high-fidelity execution environment.

This thesis aggregates and categorizes fidelity variance between virtual machines and their physical counterparts, and classifies these variances into five domains based on their unique characteristics and potential for application in high-fidelity virtualization. We further conducted an in-depth analysis on each domain to assess the challenges and practicality of implementation in a high-fidelity virtualization environment. Finally, we presented a methodology to emulate physical characteristics of virtual hardware to create a high-fidelity virtual machine.

14. SUBJECT TERMS high-fidelity virtualization, cyber physical system, virtual machine	**15. NUMBER OF PAGES** 95
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

HIGH FIDELITY VIRTUALIZATION FOR CYBER OPERATIONS

Corey D. Ingraham
Lieutenant, United States Navy
BS, Pennsylvania State University, 2013

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2019

Approved by: Alan B. Shaffer
Advisor

Gurminder Singh
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Virtualization has become ubiquitous in cyberspace over the last decade, expanding into cloud technology and embedded mobile devices. Outside of a highly sophisticated user community, consumer demand has yet to realize the need for creation of a high-fidelity virtual environment, resulting in a lack of specialized hypervisor design and a stall in the evolution of a higher level of fidelity in virtual hardware.

Most modern hypervisors rely on virtual hardware designed to meet the minimum requirements of computing, leading to a low-fidelity implementation in virtual systems when compared to their non-virtualized counterparts. High-fidelity can be achieved by effectively emulating these physical characteristics in a virtual environment that could enable further development of cyber operations by providing all the benefits of virtualization coupled with a specialized high-fidelity execution environment.

This thesis aggregates and categorizes fidelity variance between virtual machines and their physical counterparts, and classifies these variances into five domains based on their unique characteristics and potential for application in high-fidelity virtualization. We further conducted an in-depth analysis on each domain to assess the challenges and practicality of implementation in a high-fidelity virtualization environment. Finally, we presented a methodology to emulate physical characteristics of virtual hardware to create a high-fidelity virtual machine.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

BIOS	Basic Input Output System
CBD	Counter-Based Detection
COW	Copy-On-Write
CPU	Central Processing Unit
DOD	Department of Defense
EPT	Extended Page Table
GDT	Global Descriptor Table
GPU	Graphics Processing Unit
HFH	High-Fidelity Hypervisor
HFV	High-Fidelity Virtualization
HFVM	High-Fidelity Virtual Machine
IDT	Interrupt Descriptor Table
INVD	Invalidate Internal Caches
IOMMU	Input / Output Memory Management Unit
IP	Internet Protocol
ISA	Instruction Set Architecture
LDT	Local Descriptor Table
NVM	Non-Virtualized Machine
OS	Operating System
PCE	Physical Component Emulator
RAID	Redundant Array of Inexpensive Disks
RAM	Random-Access Memory
RDTSC	Read Time Stamp Counter
RSB	Return Stack Buffer
SIDT	Store Interrupt Descriptor Table
SRIOV	Single-Root Input Output Virtualization
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
VHC	Virtual Hardware Controller
VM	Virtual Machine

VMBR	Virtual Machine-Based Rootkit
VMM	Virtual Machine Monitor

ACKNOWLEDGMENTS

I would like to thank my family and friends who have given me encouragement and support without which I could not have achieved this goal. I would also like to thank Professor Shaffer and Professor Singh for their invaluable mentorship throughout the writing process.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PURPOSE

Virtualization has become ubiquitous in cyberspace over the last decade, expanding into cloud technology and embedded mobile devices. It provides many advantages across the cyber spectrum, including efficiency, isolation, flexibility, and scalability. Driven by consumer demands throughout the Department of Defense (DoD) and industry virtualization has evolved to provide a host of solutions. Outside of a highly sophisticated user community, consumer demand has yet to realize the need for creation of a high-fidelity virtual environment, resulting in a lack of specialized hypervisor design and a stall in the evolution of a higher level of fidelity in virtual hardware.

Most modern hypervisors rely on virtual hardware designed to meet the minimum requirements of computing, leading to a low-fidelity implementation in virtual systems when compared to their non-virtualized counterparts. For instance, physical sensor characteristics of virtual hardware, such as Central Processing Unit (CPU) temperature, are not emulated in most modern hypervisors. As a result, cyber operations designed to manipulate, or take advantage of, physical sensor data become ineffective in virtualized environments. For example, a cyber-attack designed to directly manipulate CPU temperature information will be rendered ineffective in a virtual environment because virtual CPUs are not emulated at a level of fidelity to support such functionality. This lack of functionality is seen to be the case for almost all hardware sensor information in virtual environments, creating significant shortcomings in virtualization application and offering a unique opportunity for the evolution of virtualization technology tailored specifically toward specialized cyber operations. Effectively emulating these physical characteristics in a virtual environment could enable further growth and development of cyber operations by providing all the benefits of virtualization coupled with a specialized high-fidelity execution environment.

B. RESEARCH OBJECTIVE

The objective of this work was to identify potential methods of creating a high-fidelity virtual environment conducive to specialized cyber operations. This research also defined an initial methodology to integrate high-fidelity virtual hardware in a VM to enable holistic emulation of physical sensor data.

C. RESEARCH QUESTIONS

We sought to achieve the objective of this thesis by bounding our research with three research questions.

1. Primary Question

What specific characteristics not currently implemented in modern virtualization technology could be included to extend a VM to create a high-fidelity virtualized environment?

2. Secondary Question

Which of these five HFV characteristics could be used to influence the behavior of a virtual machine if emulated appropriately?

3. Tertiary Question

How could one of these characteristics be emulated at a level of fidelity necessary to influence VM behavior in the same manner it would influence NVM behavior?

D. BENEFITS OF STUDY

This research will benefit the DoD by providing a methodology to develop high-fidelity virtual environments that can be implemented to create specialized VMs tailored to cyber operations. This research will also enable the cyber warfare community to develop high-fidelity target emulation for malware testing and analysis in offensive and defensive cyber operations and will be applicable in industry across multiple cyber domains to include hypervisors, virtual machines, and cyber security.

E. ORGANIZATION

This thesis is organized into four additional chapters: background, high-fidelity virtualization characteristics, high-fidelity virtualization assessment, and conclusion and future work.

The next chapter provides a taxonomy of the various virtualization domains that are currently implemented across the DoD and industry. Key terminology related to virtualization is discussed to establish a clear vocabulary for the purposes of this thesis. This chapter also discusses a high-level overview of VMs and hypervisors, and discusses the several technologies used to virtualize the CPU, memory, and I/O devices. Lastly, this chapter introduces and formally defines the concept of high-fidelity virtualization.

Chapter III outlines a taxonomy of high-fidelity virtualization characteristics that could potentially be implemented in a high-fidelity virtual environment. Several domains are defined to categorize characteristics based on their roles in virtualization technology and their potential impact on high-fidelity virtualization.

Chapter IV discusses the benefits, challenges and considerations, and potential methodologies for each high-fidelity virtualization domain outlined in the previous chapter. It assesses each domain based on the concept of creating a VM that more holistically emulates a non-virtualized machine.

The last chapter addresses overarching challenges and limitations of developing a high-fidelity hypervisor, and identifies areas that require further research and development necessary to achieve the concept of high-fidelity virtualization.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. INTRODUCTION

Before exploring the concept of high-fidelity virtualization, it is important to explore the key concepts related to virtualization technology. Virtualization is a complex concept that has been defined in many ways. This complexity is due, in large part, to the inherently different types and implementations of virtualization that exist today. Virtualization can be broadly defined as a layer of abstraction in the form of software that is inserted at various levels of computing architecture in order to achieve a multitude of computing solutions or operating environments. In order to define virtualization to an appropriate scope for this thesis, exploration of the many facets of virtualization technology must first be discussed.

B. SURVEY OF VIRTUALIZATION TECHNOLOGY

VMware provided a solution to x86 virtualization in the late 1990s resulting in a revival of virtualization technology that was previously considered impractical due to technical roadblocks. Coupled with fast-paced advancements in hardware technology, the virtualization landscape quickly blossomed, becoming rather complex and amorphous, with many different implementations being utilized across industry and the DoD. This section will explain how virtualization technology can generally be broken down into seven domains.

1. Desktop

Desktop virtualization creates a layer of abstraction between hardware and the desktop presented to the user. The "desktop" in this sense includes the operating system, applications, and user data [1]. This layer of abstraction can come in two different forms in the context of desktop virtualization: local and streaming [1].

Local desktop virtualization, also referred to as "client," can be distinguished by an execution environment for the user residing on the client machine. In this model, the virtualized desktop and the hardware required to execute that desktop are both on the same

machine [1]. VMware Workstation, VirtualBox, and Xen are examples of client desktop virtualization technology.

Streaming desktop virtualization, referred to as Virtual Desktop Infrastructure by VMware, or "server desktop virtualization," can be characterized by an execution environment for the user's virtualized desktop residing in a remote location, separate from the client machine. In this model, the virtualized desktop is executed in a remote environment and then streamed to the client via network protocols [1]. The desktop can be provided remotely by cloud-based solutions as well as remote servers. Streaming desktop virtualization via the cloud is also referred to as Desktop as a Service, where the virtual desktop is streamed to the end user via cloud service providers rather than local servers. VMware Horizon and Citrix Virtual Desktop are examples of streaming desktop virtualization technology.

2. Server

Server virtualization is perhaps most widely thought of as the "traditional" form of virtualization. Server virtualization is accomplished by placing a layer of abstraction between hardware and a fully functioning server. This layer of abstraction, in the form of software, multiplexes the hardware of the machine, allowing multiple "virtual" servers to be installed on one physical machine. The multiplexing of hardware resources in order to run multiple virtual servers on a single physical machine provides many advantages to network administrators, including more efficient power and processing utilization across a network. Additional network management-specific features further differentiate server virtualization from desktop virtualization. These features allow network administrators to quickly and easily maintain, troubleshoot, and optimize their network in ways only possible through virtualization technology. VMware ESXi and Xen Server are common forms of server virtualization technology.

3. Application

Application virtualization refers to the technique of creating a layer of abstraction between running applications on a physical machine and the operating system installed on that machine. Traditionally, applications are executed and managed directly by an

operating system. Application virtualization, however, isolates an application from the operating system utilizing one of two methods: sandboxing or streaming [1].

Sandbox application virtualization isolates an application from the operating system by packaging all the application dependencies into a single instantiation of the application. The application runs in its own execution environment, isolated from other applications concurrently running on the machine, without any dependencies being provided from the operating system. Docker containers and Java Virtual Machines are examples of sandbox application virtualization [2].

Streaming application virtualization refers to technology that allows users to run applications remotely, from a local server or the cloud, without having to install the application locally [3]. Commonly known as Software as a Service in the cloud computing industry, streaming application virtualization enables end users or administrators to avoid application installation and management. Microsoft App-V and Office365 are popular applications that utilize streaming application virtualization technology.

4. Storage

Storage virtualization refers to the technique of creating a layer of abstraction between physical storage and virtual storage [1]. Storage virtualization can be achieved by aggregation or multiplexing methods. For instance, storage virtualization is achieved in redundant array of inexpensive disks (RAID) technology by aggregating an array of disks into a single virtual disk [4]. In contrast, partitioning a hard drive into multiple, isolated blocks is a form of multiplexing storage virtualization on a local machine [3].

5. Network

Network virtualization refers to technologies that create a layer of abstraction in the network layer of the Open System Interconnection model, or the internet layer of the Transmission Control Protocol/Internet Protocol (TCP/IP) model which are responsible for routing traffic through a network. These technologies offer administration and security solutions for network administrators and end users, respectively. Typical examples of

network virtualization technology include Virtual Local Area Networks, Virtual IP, and Virtual Private Networks [2].

6. Resource

Resource virtualization refers to the techniques employed by operating systems to insert a layer of abstraction between hardware resources and the processes and applications that use them. Although often overlooked as a form of virtualization, resource virtualization is a common form of virtualization, and in one form or another, it is employed by nearly all operating systems. Effectively, the operating system serves as this layer of abstraction and "virtualizes" physical resources by various methods in order to meet the demands of all running processes on the machine. Paging files and process management are examples of resource virtualization.

7. Mobile

Mobile virtualization refers to virtualization techniques specifically applied in the domain of mobile devices, which in this context refers to modern embedded systems such as smart phones, tablets, automobiles, control devices, and consumer electronics [5]. Traditional embedded systems were narrow in scope and purpose, and came preloaded with a closed software stack from the device vendor [5]. However, modern embedded devices typically take on the functionality of the PC, running high-level, application-oriented operating systems [5]. Virtualization is highly desired in this domain for its ability to run multiple operating systems on a single device in addition to the security benefits virtualization provides through isolation [6].

C. COMPARING TERMINOLOGY

The scope of this thesis has been strictly in the context of the desktop virtualization domain discussed previously, specifically, local desktop virtualization. As the scope of virtualization narrows, the definition of what classifies as virtualization also narrows and must be refined.

1. Virtualization

Bugnion, Nieh, and Tsafrir define virtualization as "the application of the layering principle through enforced modularity, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized" [4]. Singh defines virtualization as

> a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others. [7]

These definitions describe a "layering principle" of "identical resources" and "multiple execution environments," all of which are essential aspects of virtualization technology. Combining these concepts, we define virtualization for the scope of this thesis as a layer of abstraction in the form of software that creates one or more execution environments, where the execution environment requires no cross-architectural translation, and directly executes on the CPU.

2. Emulation

The term "emulation" is often confused with virtualization, and the terms are often used interchangeably; however, they are not the same.

Bugnion, Nieh, and Tsafrir define emulation as "a level of indirection in software to expose a virtual resource or device that corresponds to a physical device, even if it is not present in the current computer system" [4]. Mallach defines emulation as "a process whereby one computer is set up to permit execution of programs written for another computer" [8]. These definitions describe the essential feature that defines emulation, i.e., that it replicates one hardware architecture on another [4]. The process of translating one CPU architecture to another introduces a significant amount of overhead not required in a virtualized machine.

Machine emulators are typically used to provide cross-architectural support for applications or for application development. QEMU and Genymotion are examples of common emulators.

3. Simulation

Simulation is another term often used in conjunction with virtualization and emulation, and must be examined to understand the relationship between the three terms.

A computer simulator "is typically implemented as a normal user-level application, with the goal of providing an accurate simulation of the virtualized architecture, and often runs at a small fraction of native speed, ranging from a 5 to a 1000 times slowdown, depending on the level of simulation detail" [4].

Since a simulator is run at the application level, it incurs a significant "processing tax" [4]. As a result, simulation is the least efficient method in terms of equivalence to a physical machine. Machine simulators are typically used for modelling and analysis of complex problems. Simulink is an example of machine simulator software.

To summarize, the predominant distinguishing factor between the three terms is determined by the method of CPU execution. Both virtualization and emulation execute on native hardware, however, they do so in different ways. Virtualization is defined by CPU execution that requires no cross-architectural translation, while emulation does require such translation. In contrast, simulation CPU execution occurs entirely in software whereby the simulated instruction set does not directly execute on the host CPU.

It is important to note that the term virtualization may still be used to describe software that utilizes emulation techniques to create a virtual environment. The nuanced differentiation between virtualization and emulation in this case is that a virtualizer will execute a predominant subset of CPU instructions without requiring the need for translation, whereas an emulator will translate many, if not all, instructions prior to CPU execution. Therefore, a virtualizer may perform translation on some instructions and still be considered a virtualizer, so long as most instructions are executed without the need for translation.

In terms of processing speed, virtualization offers the optimal solution, followed by emulation, and simulation. Even though emulation requires architectural translation, emulation still offers faster processing speeds than simulation since a simulation does not execute directly on a host CPU.

D. VIRTUAL MACHINES

Virtual machines are perhaps the most common implementations of virtualization technology. Virtual machines provide efficient solutions to many problems faced by the DoD and industry today. They can be used by network administrators to allow more efficient use of resources by consolidating multiple servers on one physical machine, or by security professionals to create an isolated and secure operating environments from which to test software [7].

Popek and Goldberg define a virtual machine as "an efficient, isolated duplicate of the real machine" [9]. A virtual machine is often referred to as the "guest," and the machine it is installed on is often referred to as the "host." This language makes it easier to describe virtual environments. Applying Popek and Goldberg's definition, the guest machine is the "isolated duplicate" of the host machine. This concept can be easily described pictorially. Figure 1 represents a Non-Virtualized Machine (NVM) while Figure 2 represents a computer implementing virtualization with two virtual guest machines installed on a host machine. We will use the term NVM to describe a traditional machine not employing virtualization techniques throughout the remainder of this thesis.

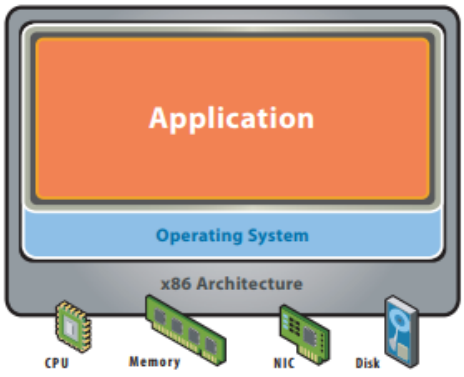


Figure 1. Non-Virtualized Machine. Source: [10].

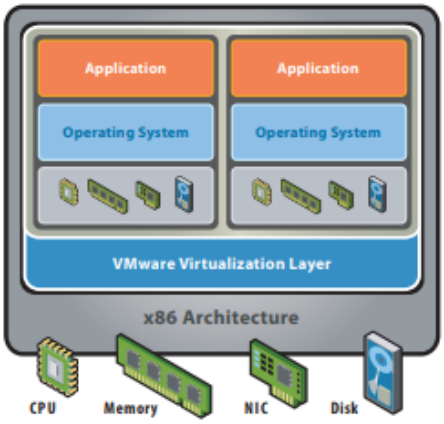


Figure 2. Virtualized Machine. Source: [10].

As Figure 2 depicts, a virtual machine is a representation of a physical machine using software. The virtualization software, illustrated as "VMware Virtualization Layer" in the figure, serves as the layer of abstraction between physical hardware on the host, and the virtual machine guests. The virtualization layer is responsible for multiplexing the host machine's physical resources into "virtual resources" which are used by the virtual machines. This multiplexing allows the virtualization layer to allocate several instantiations of virtual resources and thus create multiple virtual machines on one physical host. Creating virtual machines is not a trivial process, and several technical factors must be considered when implementing them.

Popek and Goldberg defined virtual machines in their paper *Formal Requirements for Virtualizable Third Generation Architectures* in 1974. They described three properties that must be present for a machine to be considered a "virtual machine" and thus qualify as an implementation of virtualization technology: efficiency, resource control, and equivalence [9].

The efficiency property states "all innocuous instructions are executed on the hardware directly, with no intervention at all on the part of the [virtualization layer]" [9]. In simpler terms, this property states that the virtual machine must execute the majority of its CPU instructions directly on the host CPU. This property ensures that the virtual machine processes instructions in as close as possible to the time it would take a NVM to

perform the same instruction. The efficiency property also excludes emulation from qualifying as a virtual machine, as emulators do not execute instructions directly on the host CPU.

The resource control property states "it must be impossible for [an] arbitrary program to affect the system resources" [9]. In other words, no program running in the virtual machine should be able to interact directly with the host's hardware resources without allocation of control from the virtualization layer. This property serves to enforce isolation and security of the host machine from the virtual machine.

The equivalence property states that all programs executing in the virtual machine must "perform in a manner indistinguishable" from the manner they would perform on a NVM [9]. This property, also referred to as fidelity [2], serves as a measure to verify that the virtual machine behaves in a manner as close as possible to a NVM.

These properties define what constitutes a virtual machine, but the virtual machine itself is incapable of enforcing them. The responsibility of satisfying and enforcing these three requirements lies within the virtualization layer, also known as the hypervisor.

E. HYPERVISORS

Hypervisors serve as the virtualization layer between hardware and software to create an execution environment for the virtual machine. Popek and Goldberg refer to hypervisors as "virtual machine monitors" when describing the properties of virtual machines [9]. Although the properties of efficiency, resource control, and equivalence are properties of virtual machines, they are also characteristics of hypervisors, as the hypervisor is responsible for creating an execution environment bounded by these properties.

The main role of the hypervisor is to multiplex resources of the host machine and control access to those resources for distribution across all virtual machines operating on the host, while enforcing the properties of efficiency, resource control, and equivalence. Hypervisors come in two basic types: Type I and Type II.

1. Type I Hypervisors

Type I hypervisors, also known as “bare metal” hypervisors, are installed directly on hardware. Figure 3 illustrates a Type I hypervisor.

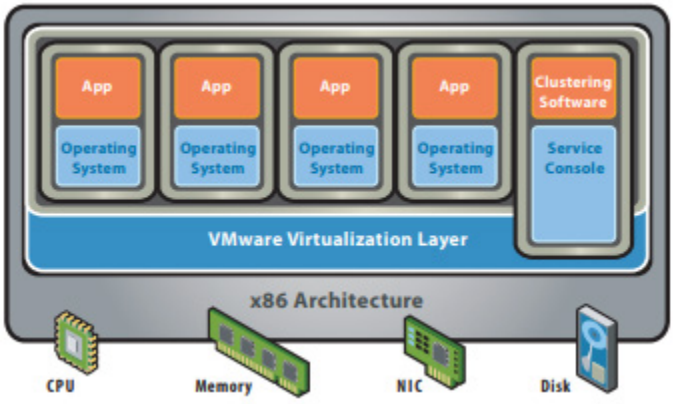


Figure 3. Type I Hypervisor. Source: [10].

Type I hypervisors are distinguished by requiring no operating system on the host machine. The hypervisor serves as both an operating system and a virtual machine monitor simultaneously. Type I hypervisors have full control of the host machine and do not need to coordinate resource allocation with an operating system. VMware ESXi and Xen are examples of Type I hypervisors.

2. Type II Hypervisors

Type II hypervisors run as an application “on top” of a host machine operating system. Figure 4 illustrates a Type II hypervisor.

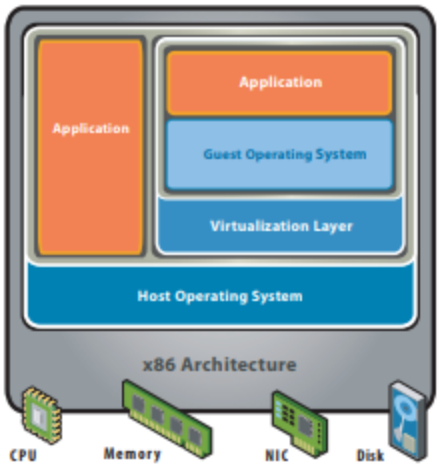


Figure 4. Type II Hypervisor. Source: [10].

Type II hypervisors are a unique type of application and interact with host hardware differently than a typical application. Type II hypervisors have full control of the host CPU while executing the guest operating system [4]. This process is balanced between the hypervisor and the host operating system utilizing a "world switch," similar in function to a CPU context switch [4]. An example of a Type II hypervisor is VMware Workstation or VirtualBox.

F. VIRTUALIZATION TECHNIQUES

The virtualization industry has developed many techniques for creating hypervisors, each with its own advantages and disadvantages. The hypervisor is responsible for virtualizing three main hardware components to create a virtual machine: CPU, memory, and input/output devices.

1. CPU Virtualization

Perhaps the most important decision when developing a hypervisor is the method in which the CPU will be virtualized. The method chosen is dependent on the Instruction Set Architecture (ISA) of the CPU, as each architecture is different and requires a hypervisor tailored to the specifics of how that ISA executes instructions. Due to its wide use by consumers, Intel's x86 is the most commonly virtualized ISA [11]. The x86

instruction set was originally thought to be not "classically virtualizable" due in part to the way x86 handles privileged instructions run in user mode [12]. "Classically virtualizable" refers to the ability to virtualize an architecture solely using the trap-and-emulate method, explained below [12]. However, in 1998 VMware developed a method to virtualize x86 architecture utilizing binary translation [13]. This thesis focuses strictly on Intel x86 architecture virtualization.

a. Intel x86 Privilege Levels

Intel x86 processors run natively in protected mode, which is used to establish privilege levels necessary to separate kernel execution from user mode execution [4]. Figure 5 illustrates the Intel x86 privilege levels, also referred to as rings. Ring 3 represents the least privileged ring and is allocated for user mode operations. Ring 0 represents the ring with maximum privilege and is allocated for kernel mode operations. User applications normally execute in Ring 3 while the computer operating system runs in Ring 0 [13].

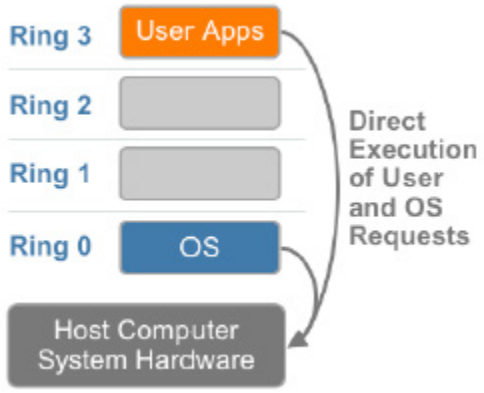


Figure 5. Intel x86 Privilege Levels. Source: [13].

b. Trap-and-Emulate

Under the trap-and-emulate paradigm, all privileged instructions from the virtual machine are "trapped" by hypervisor to prevent executing guest privileged instructions on the host CPU. Once the instruction is trapped, the hypervisor emulates the result of the privileged instruction in the virtual environment as if it had been executed on the CPU

directly [11]. This process effectively isolates the virtual machine and prevents it from executing privileged instructions in the host execution environment, which would pose a significant security risk.

Trap-and-emulate alone was not sufficient to virtualize x86 architecture, however, due in part to the inability to force all privileged instructions to trap to the hypervisor [11]. To solve this problem, VMware introduced binary translation [13].

c. Binary Translation

Binary translation is an efficient form of emulation [4] and is only utilized when executing guest kernel mode instructions [11]. All user mode instructions executed by the guest are directly executed on the host CPU and are not passed to the binary translator. When the guest attempts to execute kernel mode instructions, they are intercepted by the binary translator and grouped into blocks. The translator then processes the blocks and translates only the privileged instructions into unprivileged, user mode instructions. Once a block is translated, it is cached in order to prevent future translations. Over time, a decreasing number of translations occur as the translator stores the working set of the virtual machine [11]. VMware utilizes binary translation and full virtualization to provide virtualization solutions to consumers [13].

d. Full Virtualization

In a full virtualization model, the guest operating system is completely unaware that it is operating in a virtual environment. The guest operating system is unaltered and can be installed on the virtual machine directly from the manufacturer. The guest OS will request to execute privileged instructions identically to the way it would if it were installed on a NVM. The hypervisor is responsible for catching those privileged instructions, using either trap-and-emulate or binary translation techniques. VMware utilizes binary translation in their full virtualization model. Figure 6 illustrates an implementation of full virtualization utilizing binary translation.

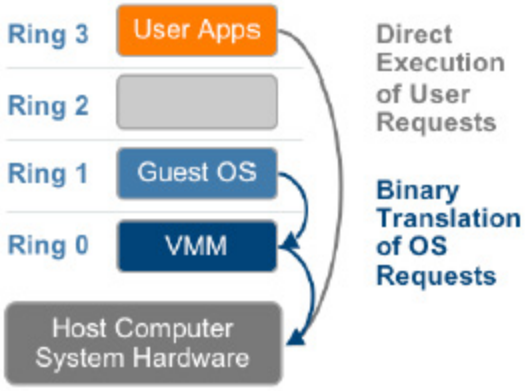


Figure 6. Intel Full Virtualization with Binary Translation CPU Execution. Source: [13].

The hypervisor resides in Ring 0 in full virtualization while the guest OS resides in Ring 1. All guest user applications reside in Ring 3 and will directly execute on the host CPU. When the guest OS makes instruction calls from Ring 1, the hypervisor performs binary translation and runs the translated binary on the host CPU. Full virtualization is valued by consumers due to its portability and ease of use, because it does not require operating system modifications.

e. Paravirtualization

Paravirtualization, commonly referred to as Operating System Assisted Virtualization [13], was first widely implemented by Xen. Unlike full virtualization, paravirtualized operating systems must be modified to create "hypercalls" [14]. Hypercalls, similar to traditional system calls, are utilized by the guest operating system to communicate directly with the hypervisor, and consist of rewritten privileged instructions that can be directly executed on the host CPU [4]. The Xen paravirtualized hypervisor operates using only direct execution, and removes the need for the hypervisor to perform trap-and-emulate or binary translation techniques [14]. Unlike full virtualization, the guest operating system recognizes it is operating in a virtual machine since it is communicating directly with the hypervisor. The Xen hypervisor operates in Ring 0 while the guest OS operates in Ring 1 in the same manner as full virtualization [4]. Paravirtualization offers a

significant performance advantage in many cases due to the reduced overhead as a result of not requiring trap-and-emulate or binary translation methods [14].

f. Hardware-Assisted CPU Virtualization

Intel and AMD implemented first-generation hardware support for virtualization in the form of VT-x and AMD-V, respectively. This first-generation support was designed primarily to improve CPU virtualization by introducing a new method of virtualization that does not require paravirtualization or binary translation [4].

Intel VT-x introduced an additional "root mode" of execution wherein both the hypervisor and the host operating system operate, while the guest machine operates in "non-root" mode [4]. Root mode was designed to only be utilized for virtualization purposes and the CPU can transition between root mode and non-root mode based on single instructions, removing the need for a complex set of instructions prior to a transition between modes [4]. The transition from root to non-root mode is known as a VM Entry on the Intel CPU, while the transition from non-root mode to root mode is known as a VM Exit [15]. Non-root mode was designed to allow the guest machine to execute privileged instructions directly on the processor without requiring a trap from the hypervisor. Since non-root mode is a duplicate of the root mode architecture, any privileged instructions executed in non-root mode only have an effect in the guest machine and do not influence host machine behavior. Traps are still required in some circumstances, however, such as when the CPU transitions from non-root mode to root mode, referred to as an "exit" [11]. First-generation hardware support for virtualization eliminated the need for binary translation [16]. Figure 7 depicts Intel VT-x privilege rings.

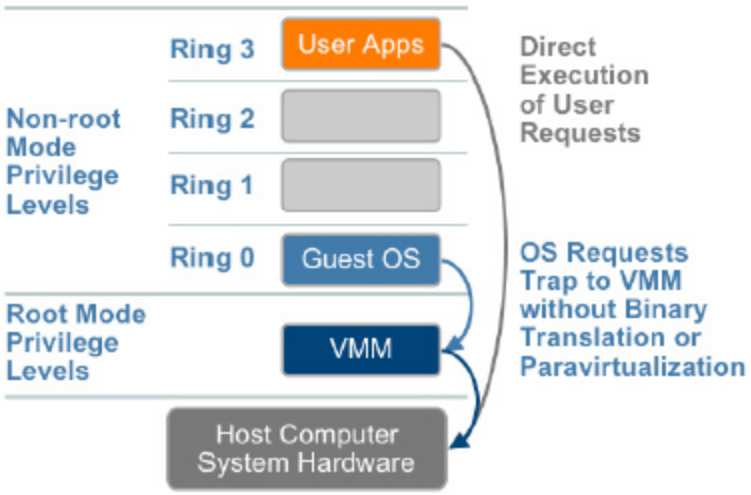


Figure 7. Intel Hardware-Assisted CPU Execution. Source: [13].

2. Memory Virtualization

Managing memory allocation between a host-physical machine and a guest-virtual machine is a very crucial process in virtualization. Allowing either the guest or host to write or access the other's memory space could cause fatal system errors. An important distinction in memory virtualization is the difference between guest-virtual memory and guest-physical memory. From the perspective of the host, both are considered "virtual" memory and must be mapped to the host-physical memory. However, from the perspective of the guest, virtual and physical memory are equivalent to virtual and physical memory in any NVM, and function in the same manner as managed by the operating system. Figure 8 helps illustrate the difference between these layers of abstraction; "machine memory" in this figure is also referred to as host-physical memory.

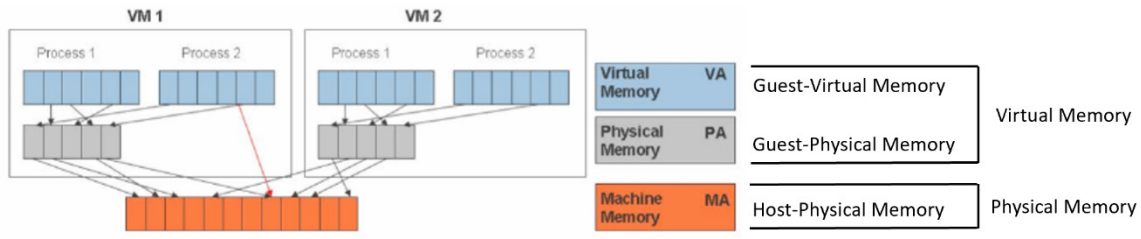


Figure 8. Memory Virtualization. Source: [13].

a. Memory Virtualization without Hardware Support

The initial method of memory virtualization prior to the introduction of hardware support was through the use of shadow paging [4]. Shadow paging is a technique used in full virtualization whereby the hypervisor maintains a software-defined shadow page table mapping guest-virtual memory to host-physical memory [4], similar to the way in which an operating system maintains a page table to map virtual memory to physical memory locations. A shadow page table is used as an additional layer of abstraction between memory mapping and can cause large overhead due to the added layer of complexity for page table lookups. Shadow paging became inefficient with the arrival of hardware support for memory virtualization [4].

b. Memory Virtualization with Hardware Support

Second-generation hardware support for memory virtualization came in the form of Extended Page Tables in the case of Intel, and Rapid Virtualization Indexing in the case of AMD [11]. Although different in name, both technologies behave in a similar manner and are also referred to more broadly as Nested Page Tables, or nested paging [11]. Nested paging introduced a method wherein the hypervisor is not required to manage software-defined shadow page tables [4]. To accomplish this technique, the traditional hardware-defined page table used by the operating system is combined with the software-defined page table used by the hypervisor using strictly hardware [4]. Nested paging allows the use of one combined page table for both the host operating system and the guest virtual machine, removing the overhead accumulated in shadow paging from performing two Translation Lookaside Buffer (TLB) mappings to resolve virtual-guest addresses to host-physical addresses. Nested page tables dynamically work in two different modes to allow virtual-guest addresses to be mapped directly to host-physical addresses [11].

3. Input / Output Device Virtualization

I/O devices comprise the third category of hardware components that must be virtualized to create a virtual machine. A hypervisor must implement a method to virtualize physical I/O devices in order to account for both the guest and host OS attempting to access

these devices. Assuming no hardware support for virtualization, there are two main methods of I/O virtualization, delineated by full virtualization and paravirtualization.

In a full virtualization model, the guest OS is unaware it is being virtualized and therefore expects to have full control over the I/O devices under its purview. To account for this, the hypervisor "interposes" on the guest machine to create virtual I/O devices that are not directly connected to physical host devices [4]. This interposition allows the hypervisor to trap all requests from the guest machine to I/O devices, and emulate their responses in the guest machine, decoupling and recoupling virtual I/O to physical I/O as necessary [4]. This provides isolation between the virtual and physical I/O devices through a level of indirection in software.

Paravirtualization affords a different approach to I/O virtualization. A paravirtualized OS implements a standardized set of device drivers designed to work with virtualization in mind. These guest device drivers are only compatible with the hypervisor and cannot interface with any physical devices directly. In some cases, paravirtualized I/O offers performance increases but at the cost of requiring modification to the guest OS [4].

The introduction of hardware support for I/O virtualization eliminated the need for interposition in the form of trap-and-emulate as well as paravirtualization with respect to I/O devices [4]. Hardware support introduced a new method called direct device assignment, which gives the guest OS direct and exclusive control of the physical I/O device. Direct device assignment reduces virtualization overhead but raises two main concerns: security and scalability [4]. The ability of the guest machine to access and write to physical host I/O memory poses a significant security concern. In addition, scalability poses a problem, as the number of physical I/O devices installed on a host machine is much smaller than the number of potential guest machines that could be operating the same host machine, causing a bottleneck in allocation. Both of these problems are solved by hardware support via the I/O Memory Management Unit (IOMMU) and Single-Root I/O Virtualization (SRIOV) [4].

The IOMMU adds a layer of abstraction via hardware in order to provide security between the guest and host machine. Rather than allowing the guest OS to make direct

memory accesses and directly receive interrupts to/from physical I/O devices, the IOMMU acts as an intermediary, translating virtual memory accesses/interrupt vectors to physical addresses/interrupts vectors via lookup tables defined in the hypervisor [4]. This process effectively isolates the guest and host machine while still allowing the guest to have "direct" access to the I/O device [4].

SRIOV solves the scalability problem by introducing I/O devices capable of multiplexing themselves to software. Traditionally, the OS is responsible for multiplexing I/O devices; however, SRIOV devices are capable of presenting multiple virtual instances of itself to the OS that can each be directly assigned to a guest machine [4]. SRIOV devices efficiently multiplex themselves at the hardware level, allowing guest machines to access them directly with no intervention or management from the hypervisor [4].

E. HIGH-FIDELITY VIRTUALIZATION

High-Fidelity Virtualization (HFV) is a new domain of virtualization technology that focuses on improving upon the equivalence property of virtual machines defined by Popek and Goldberg. The principle idea behind HFV is to improve upon an existing hypervisor in order to facilitate the creation of a virtual machine that more closely captures the holistic operation of a NVM.

As mentioned earlier in this chapter, virtualization in this context is not to be confused with emulation. There has been much work in the field of high-fidelity emulation of many different platforms, which can be very useful in application development, where new applications can be tested in a holistic environment. Genymotion, for example, is a high-fidelity android emulator that emulates battery, Global Positioning System data, accelerometer, network connectivity, and multi-touch functionality of smartphones on a desktop or in the cloud [17]. This software qualifies as a high-fidelity emulator due to the level of equivalence it provides in emulating physical device characteristics and signals in software. HFV seeks to achieve the same goal, but through virtualization rather than emulation.

In a research paper titled *High Fidelity Virtualization of Cyber-Physical Systems,* Zhang, Xie, Dong, Yang, and Zhou introduced a novel approach to synchronize a virtual

machine with a physical component emulator [18]. The focus of this research was to enable "real software, virtual hardware, and virtual physical components to execute in a holistic virtual execution environment." In this research, the term virtualization is used to describe the holistic virtual environment created by the synchronization of an emulated virtual machine and a physical component emulator [18].

Zhang et al. performed two experiments to demonstrate the integration of cyber and physical components in a holistic virtual environment. They developed a physical component emulator using MATLAB to emulate physical components such as sensors, actuators, and control plants and utilized a QEMU virtual machine to emulate a cyber-physical system software controller [18].

In the first experiment, Zhang et al. created a virtual execution environment for a TableSat cyber-physical system. TableSat is a system used to "emulate the dynamics, sensing, and actuation capabilities required for satellite attitude control" [18]. Zhang et al. virtualized the TableSat using MATLAB/Simulink to emulate physical components such as dynamics, friction, fans, and sensors and used QEMU to virtualize the TableSat software controller [18]. They then developed a synchronization protocol to create a synchronized communication channel between QEMU and MATLAB, resulting in a holistic virtualization of the TableSat system. They tested and compared the virtual TableSat performance against the real TableSat performance and measured the divergence between the two systems. This experiment demonstrated that the virtual TableSat could simulate the real TableSat with reasonable accuracy [18]. The goal of this experiment was to show that it is possible to enable development of software in a virtual environment before the real physical environment becomes available [18].

In the second experiment, Zhang et al. used a similar approach to create a holistic virtual environment for an automatic transmission. They used QEMU to emulate the automatic transmission controller and MATLAB/Simulink to emulate physical components of the transmission. They demonstrated that their virtual transmission achieved a higher level of fidelity when compared to a Simulink Stateflow simulation for the same transmission [18].

The research conducted by Zhang et al. was primarily conducted to demonstrate that cyber-physical system software controllers can be tested and developed before physical components are available by utilizing virtualization. Although their research was designed for cyber-physical systems, it could have potential applications in the field of HFV.

Due to Goldberg's law of resource control to define virtual machines, a virtual machine must be safely isolated from host hardware in order to prevent unintended security breaches. This inherently limits a virtual machine, a virtualized desktop computer in this case, from achieving the same level of fidelity as a NVM. A NVM can interact with hardware in ways that a virtual machine cannot achieve, however, using the synchronization methodology presented by Zhang et al., a new avenue of achieving a higher level of fidelity in virtual machines is possible by synchronizing emulated hardware with the virtual machine while still obeying the law of resource control outlined by Goldberg.

F. SUMMARY

This chapter provided a review of the current domains in virtualization technology. This review was conducted to appropriately scope this thesis to the desktop virtualization domain. Within the realm of desktop virtualization, we defined and differentiated the terms virtualization, emulation, and simulation. This chapter further defined virtual machines and hypervisors and discussed their basic operation and interactions. We also examined virtualization techniques both with, and without, hardware support for virtualization providing a general overview of how the CPU, memory, and I/O devices are virtualized. Lastly, we defined HFV and compared this idea to current technologies and research in the field of high-fidelity emulation.

The next chapter will analyze a taxonomy of virtualization characteristics that have potential to be implemented in HFV.

THIS PAGE INTENTIONALLY LEFT BLANK

III. HIGH-FIDELITY VIRTUALIZATION CHARACTERISTICS

A. INTRODUCTION

The main body of research that contributes to identification of HFV characteristic is in the field of hypervisor detection. This field originated in 2007 after the introduction of two hypervisor rootkits in 2006, Blue Pill and Vitriol, released by Joanna Rutkowska, a researcher in the field of computer security engineering, and Matasano Security Labs, respectively [19]. Later classified as Virtual Machine Based Rootkits (VMBRs), these rootkits were designed to be an undetectable form of malware that utilized a hypervisor to hide their presence [20]. VMBRs "install a virtual-machine monitor underneath an existing operating system and hoists the original operating system into a virtual machine" [20]. This process allows the newly installed "ultra-thin hypervisor" to intercept all system calls and interrupts in order to hide its presence and discreetly exfiltrate data from the machine without triggering any alarm to the native operating system [21].

Following the development of VMBRs, the desire to detect these rootkits, and thereby detect the presence of hypervisors, became an immediate security goal. As a result of this goal, significant research was conducted in the field of hypervisor detection, revealing many characteristics that could be used to identify the presence of virtualization. These identifying characteristics have an inherently close relationship to the field of HFV. Since HFV seeks to present a virtual machine that more holistically resembles a NVM, characteristics that can be used to detect hypervisors can also be used in implementing High-Fidelity Virtual Machines (HFVMs). A virtual machine that is harder to detect (and hence, is closer to being undistinguishable from a NVM) is to a large degree inherently a HFVM. The more of these characteristics that can be implemented in HFV, the higher the level of fidelity.

This chapter examines the differences between non-virtual and virtual machines, and establishes a taxonomy of characteristics with respect to HFV. Furthermore, this chapter describes in depth these hypervisor characteristics through the lens of potential implementation in HFV.

B. HFV CHARACTERISTIC DOMAINS

We have organized HFV characteristics into five domains: artifacts, behavior, performance, security, and functionality.

1. Artifacts

This domain refers to virtual machine artifacts that can be attributed to the footprint left behind by the hypervisor. This can include a wide range of characteristics, largely dependent on the hypervisor chosen and the operating system being virtualized. All hypervisors are unique in the way they provide a virtual environment and, therefore, each establishes unique footprints that can further vary based upon the virtualized operating system. This discussion of artifacts is not exhaustive, but rather serves to demonstrate the most common artifacts left behind by popular commercial hypervisors. We have organized these artifacts into three categories based on research by Liston and Skoudis: processes, file systems, and registry; virtual hardware; and memory [22].

a. Processes, File Systems, and Registry

Virtualization artifacts can most commonly be found in the virtual machine processes, file systems, or registry. Most commercial hypervisors were not designed to avoid detection. As a result, many references to the hypervisor manufacturer are "imprinted" on the virtual machine when it is created. For example, in the case of VMware installed on Windows, over 50 references to "VMware" or "vmx" can be found in the file system, and over 300 references can be found in the registry [22]. On Linux, the /proc and /sys folders contain numerous references to the hypervisor on a virtualized system [22]. Many other Linux commands can be used to reveal the presence of these artifacts such as "dmidecode," "dmesg," "lshw," and "lspci," to name a few [23]. These artifacts can vary in number and location based on the specific hypervisor and operating system being analyzed, but generally speaking, hypervisor artifacts are typically easy to detect.

b. Virtual Hardware

We can also examine for static artifacts left behind by the hypervisor when creating and implementing virtual hardware. We have defined two sub-domains of virtual hardware: unique identification and resource allocation.

(1) Unique Identification

Virtual hardware can often be uniquely identified based on artifacts imprinted on the VM hardware by the hypervisor. For example, the virtual Network Interface Card assigned in VMware contains a Media Access Control address with a VMware manufacturer code [22]. In addition, the virtual Universal Serial Bus controller and Small Computer System Interface controller implemented in VMs can be uniquely identified by virtual hardware identifiers [22].

A similar method was introduced at Black Hat Asia in 2014 by Li et al. [24]. They conducted research to demonstrate that a VM could be detected by examining its display properties, utilizing a Java web script to return the number of resolutions supported by a target machine. Testing their method on a NVM and VMs running in VMware and VirtualBox, they observed that the host machine supported 38 resolutions while the VMs supported far fewer: 25 in the case of VMware and 4 in the case of VirtualBox [24]. In addition, they noted that VMware supported a very rare screen resolution, 1041x1041, not commonly found on NVMs because odd numbers are not typically used for resolution size [24].

(2) Resource Allocation

Many VMs can be identified heuristically by simply examining the allocation of hardware resources. VMs are usually allocated far fewer resources than even modest-sized computers, on the order of 20–100 GB of hard drive space and 2 GB of Random-Access Memory (RAM). While these artifacts alone cannot be used to identify a VM with absolute certainty, they can be an indicator of likely hypervisor detection, supporting other detection domains or characteristics.

c. Memory

Virtual machines run on top of host software, which can be a bare metal hypervisor (Type I), or a host operating system configuration (Type II). Either way, the VM cannot share the same memory space as the underlying host software for security reasons. This attribute of virtualization can lead to methods of hypervisor identification through memory, as described below.

(1) Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is an array of eight byte interrupt descriptors used by the CPU to handle interrupts in protected mode [15]. The IDT is referenced by a register (IDTR) which holds a pointer to the IDT in memory [15]. Joanna Rutkowska first introduced the concept of detecting a virtual machine via the IDTR in a program called The Red Pill in 2004 [22]. The Red Pill works by simply running the Store Interrupt Descriptor Table (SIDT) instruction, which stores the IDTR in memory, and examining where the IDT is located [22]. Rutkowska observed that on NVMs the IDT is located lower in memory while on virtual machines it is located higher in memory. Specifically, the IDT is typically stored around 0x80ffffff in Windows and 0xc0ffffff in Linux [22]. In contrast, the IDT is stored around 0xffXXXXXX in VMware VMs and at 0xe8XXXXXX in VirtualPC VMs [22]. The Red Pill simply calls the SIDT instruction and if the IDTR points to a location in memory greater than 0xd0XXXXXX, then a hypervisor is likely present [22].

(2) Global Descriptor Table

The Global Descriptor Table (GDT) is used by the processor to track memory segments using a register, the GDTR, as a pointer to the GDT [25]. The GDT can be used to detect a VM in the same manner as the IDT. Tobias Klein developed a VM detection tool called “Scoopy” that checks the value of the GDTR [22]. Similar to the IDTR, Klein observed that the GDTR is typically stored lower in memory on a NVM and higher in memory on a virtual machine [22]. Specifically, on a NVM the GDTR is less than 0xc0XXXXXX and above 0xc0XXXXXX on a virtual machine [22].

(3) Local Descriptor Table

The Local Descriptor Table is also a table of memory segments tracked by the processor, similar to the GDT, except the LDT is used for specific processes [25]. Klein also observed that the LDT can be used to identify the presence of a hypervisor in the same manner as the IDT and GDT [22]. He noticed that the LDT is typically stored at 0x00000000 on a NVM, but somewhere else in memory on a virtual machine [22].

2. Behavior

The behavior domain consists of identifiable traits inherent to the behavioral profile of virtualization technology. As opposed to static artifacts, the behavior domain measures how virtualization technology reacts to various instructions and operations, and compares that reaction to how a NVM would react under the same circumstances. Behavioral characteristics must be directly or indirectly witnessed and reflect momentary differences between non-virtual and virtual machines. We have defined two sub-categories for the behavior domain: CPU behavior and hypervisor behavior.

a. CPU Behavior

Hardware support for virtualization introduced many new CPU features to facilitate virtualization technology. Intel and AMD took similar, but unique approaches to introducing their implementation of hardware support. These nuanced approaches resulted in unique CPU behavior when handling various virtualization operations. The three examples below represent how CPU behavior can be used to identify a virtualization profile.

(1) Translation Lookaside Buffer

The TLB is a cache of recent memory address translations maintained by the operating system kernel [25]. While the page table contains a list of all physical-to-virtual address translations, the TLB is a cache of recently used physical to virtual memory resolutions. The operating system first checks the TLB for the address translation it needs when performing memory access. If the address is not in the TLB, the operating system

must perform a “page walk” and check the entire page table until it finds the address mapping it needs, resulting in much longer memory access times [25].

CPUs with virtualization support exhibit unique TLB behavior when conducting VM Exits. Intel VT-x technology flushes the entire TLB upon every VM Exit while AMD-V technology modifies one TLB entry during every VM Exit [25]. These TLB modifications introduce a detection method whereby all entries of the TLB are filled, a VM Exit is forced, and modifications to the TLB are measured [25]. If any TLB modifications are detected, this indicates virtualization technology is being implemented [26]. Brengel, Backes, and Rossow [28] further demonstrated this method to detect hardware-assisted virtualization in their research [27].

(2) Cache Invalidation

Cache invalidation was first introduced at a Black Hat conference by Ptacek, Lawson, and Ferrie [28], and relies on the Invalidate Internal Caches instruction (INVD). This instruction invalidates, or “flushes,” all CPU cache before the contents can be written to memory [15]. The cache invalidation method reveals the presence of a hypervisor by writing a pattern to memory, resulting in the write being queued in the cache, immediately followed by a forced VM Exit and an INVD instruction to flush the contents of cache [28]. If the pattern previously written to memory persists after the INVD instruction, a hypervisor is likely present [28].

(3) Return Stack Buffer

The Return Stack Buffer (RSB) is a processor structure used for speculative execution. The CPU places the return addresses of the most recent call functions onto the RSB [15]. This process creates a cache of return addresses, which speeds up function times by preventing retrieval of the return address from main memory upon function exit [15].

When a VM Exit is induced to a hypervisor, the hypervisor modifies entries in the RSB [19]. Furthermore, Maisuradze and Rossow [29] demonstrated that it is possible to detect changes in the RSB cache using several side-channel techniques. Therefore, the presence of a hypervisor can be detected by filling the RSB, forcing a VM Exit, and,

utilizing the methods proposed by Maisuradze and Rossow, detect any modifications that may have occurred in the RSB. RSB changes will indicate the presence of a hypervisor.

b. Hypervisor Behavior

Hypervisors must often behave in a unique manner in order to handle normal guest OS requests due to the inherent nature of interposition. The hypervisor must ensure the rules of efficiency, equivalence, and safety are adhered to in order to maintain its qualification as a virtual machine monitor. As such, it must interpose on virtual machines to carry out these obligations. In doing so, it reveals unique behavior that can be observed and utilized to verify its presence.

(1) IP Timestamps

Researchers from Tokyo University of Technology have demonstrated that hypervisors can be identified by IP timestamping behavior [30], [31], [32]. Noorafiza et al. demonstrated this by exploiting the IP timestamp request option in IP packet headers. The IP timestamp is an option that can be implemented in the header of IP packets to measure packet delays across a network [33]. The 32-bit timestamp is indicative of the time in the target machine, represented in milliseconds [33].

In their first experiment, Noorafiza et al. identified the presence of a hypervisor by observing a unique deviation in timestamps over a large sample size [30]. They sent repetitive timestamp request packets to three target servers on a test network and captured the timestamp replies from these three servers. One server had no hypervisor installed, thus representing a NVM, while the other two servers both had hypervisors installed, one with VMware and one with VirtualBox. Both the VMware and VirtualBox servers had a VM running that served as the target of the IP timestamp requests in the experiment. The results of the timestamp replies are shown in Figure 9.

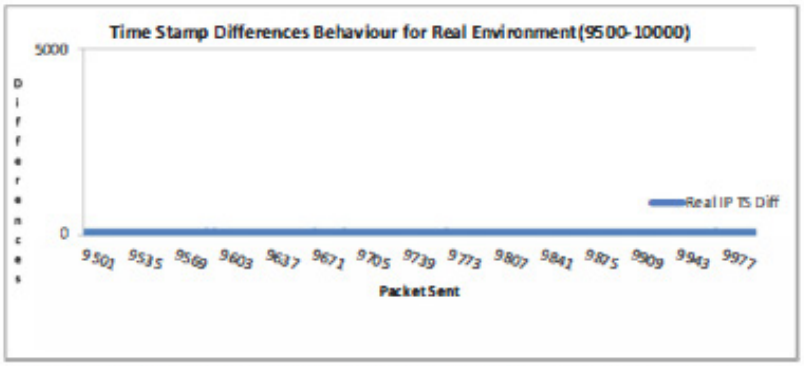


Figure 4. Timestamps differences for n+1 and n times packet reply received for real machine (n=9500 – 10,000)

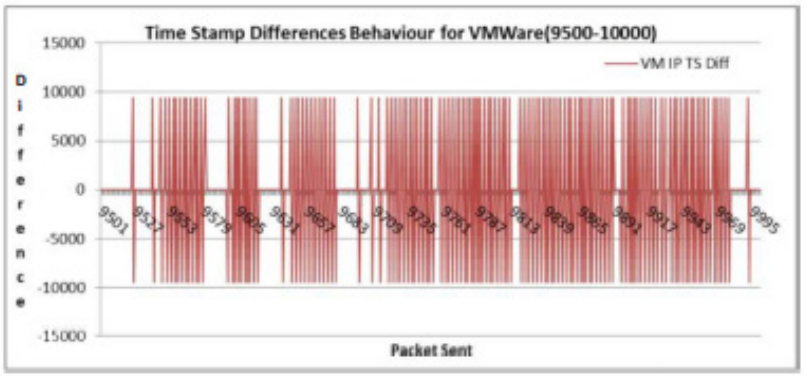


Figure 5. Timestamps differences for n+1 and n times packet reply received for VMware virtual machine (n=9500 – 10,000)

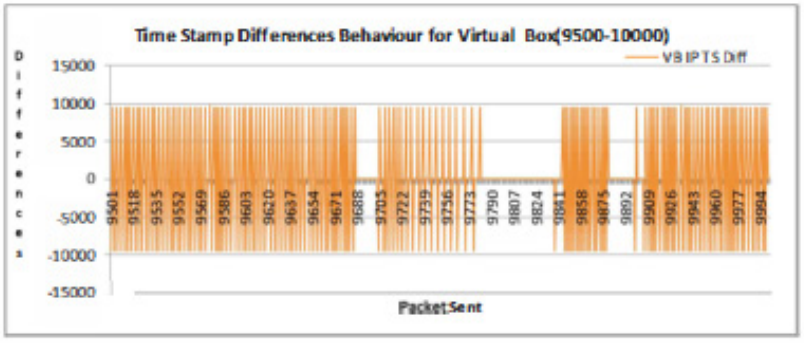


Figure 9. IP Timestamp Deviation Behavior between a NVM, VMware VM, and VirtualBox VM. Source: [30].

Figure 9 illustrates that NVMs have a very small timestamp deviation compared to virtualized machines. This method can not only differentiate a NVM from a virtualized machine, but can also differentiate between hypervisor vendors based on their unique behavior [30].

In their latest experiment, Noorafiza et al. demonstrated that hypervisors can also be fingerprinted by comparing the mean repetition values of timestamps returned by the hypervisor [31], [32]. In this experiment, they observed the number of times a target used the same timestamp rather than analyzing timestamp deviation in their prior research. They tested a NVM and three virtualized machines running three different hypervisors: XenServer, VirtualBox, and VMware [32]. Figure 10 illustrates the IP timestamp repetition behavior of these four machines.

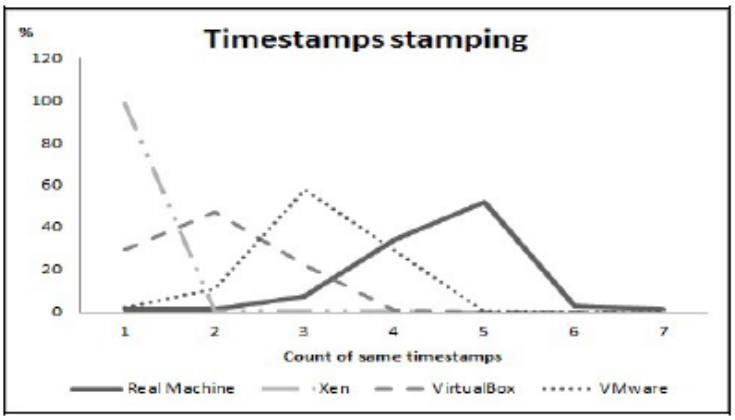


Figure 10. IP Timestamp Repetition Behavior between a Non-Virtualized Machine, XenServer VM, VirtualBox VM, and VMware VM. Source: [32].

Figure 10 shows that the machines are distinguishable from one another by their IP timestamp repetition count. The NVM used the same timestamp the most, leading to a mean repetition number of 4.5, while the hypervisors had much fewer repetitions, resulting in smaller mean repetition values of 1.03, 1.95, and 3.15 for XenServer, VirtualBox, and VMware respectively [32].

(2) Hypervisor Communication Channel

Most commercial hypervisors today implement a form of guest-to-host communication channel in order to facilitate ease-of-use operations such as a shared clipboard. This communication channel is meant to improve the quality of service for VM end users. Although this communication channel offers a level of convenience, it also

serves as a blatant indicator that a hypervisor is present. Hypervisors will typically implement a guest-to-host communication channel by either subverting current x86 instructions or creating their own proprietary, non-standard x86 instructions [22]. These instructions can be easily observed through hypervisor behavior and can, therefore, confirm the presence of virtualization [22].

3. Performance

The performance domain is comprised of methods used to determine the presence of virtualization based on machine performance. The performance domain is unique from the artifact domain in that it is not hypervisor dependent, but rather hardware dependent [34]. This domain seeks to reveal a hypervisor by taking advantage of the inherent performance differences between VMs and NVMs. We have defined three sub-categories for the performance domain: timing, counter-based, and graphical.

a. Timing

The timing variance between VMs and NVMs is one of two exceptions to the equivalence property of hypervisors stated by Popek and Goldberg [34]. This exception arises from the requirement for hypervisors to maintain control of system resources and prevent the guest machine from affecting those resources, resulting in certain instruction sequences taking longer to execute due to the interposition of the hypervisor [35]. The added layer of abstraction that a hypervisor provides inherently induces a "timing tax" on many system operations. Timing detection methods seek to reveal the presence of a hypervisor by identifying these timing discrepancies.

(1) Local Timing

The most basic form of timing detection utilizes a local timing source such as the Read Time Stamp Counter (RDTSC) instruction, which measures the amount of CPU cycles that have occurred since the last processor reset [15]. A hypervisor can be detected by timing CPU instructions that cause unconditional interception by the hypervisor and comparing the results to the time it takes to perform a benign instruction that requires no hypervisor interception [36]. Since hypervisor interception causes a VM Exit, longer

execution times will be observed for privileged instructions due to overhead induced by the VM Exit.

Although the RDTSC instruction is given in this example, any accurate timing source on the local machine can be used with this method. Ptacek, Lawson, and Ferrie mention several alternatives such as High Precision Event Timer, Local Advanced Programmable Interrupt Timer, Programmable Interrupt Timer, and Advanced Configuration and Power Interface timers [28].

(2) Translation Lookaside Buffer

Although the TLB was mentioned as a component of the behavior domain in the previous section, it can also be used to detect a hypervisor via timing methods. The TLB is required to be repopulated following every VM Exit due to the TLB flush that occurs. This process results in a timing delay, which would not have occurred on a NVM. This timing overhead can be directly measured by using a similar process to observe TLB behavior in the previous section.

This timing overhead can be detected by first reading the contents of a memory location in order to ensure an entry for that memory location is placed in the TLB. Second, measure the amount of time it takes to read that memory location again. Since the address for the memory location is in the TLB, this timing measurement will indicate how long it takes for the machine to reference the TLB. Third, force a VM Exit, which results in a TLB flush on a virtualized machine and no modification of the TLB on a NVM. Lastly, measure the time it takes to read the same memory location again [19].

If the machine is virtualized, the TLB will have been flushed by the VM Exit resulting in a necessary page walk to carry out the memory translation. The page walk will result in longer memory access times which will contrast to the operation occurring on a NVM, requiring no page walk as the memory translation will still be present in the TLB.

(3) Return Stack Buffer

The RSB can be used to measure timing differences associated with hypervisors in a similar manner to the TLB. This process directly measures the timing overhead caused

by the hypervisor's modification of the RSB during a VM Exit. To achieve this, the RSB is populated with enough nested function calls to fill the entire RSB [26]. Next, a VM Exit is induced, resulting in some previous RSB entries being evicted by the hypervisor [19]. These evictions result in RSB "misses" to occur following the VM Exit while the nested function call is being performed [26]. The overhead associated with these RSB misses can be measured and compared to the same process without inducing a VM Exit. If a hypervisor is present, a timing variation will be observed between these two procedures [26].

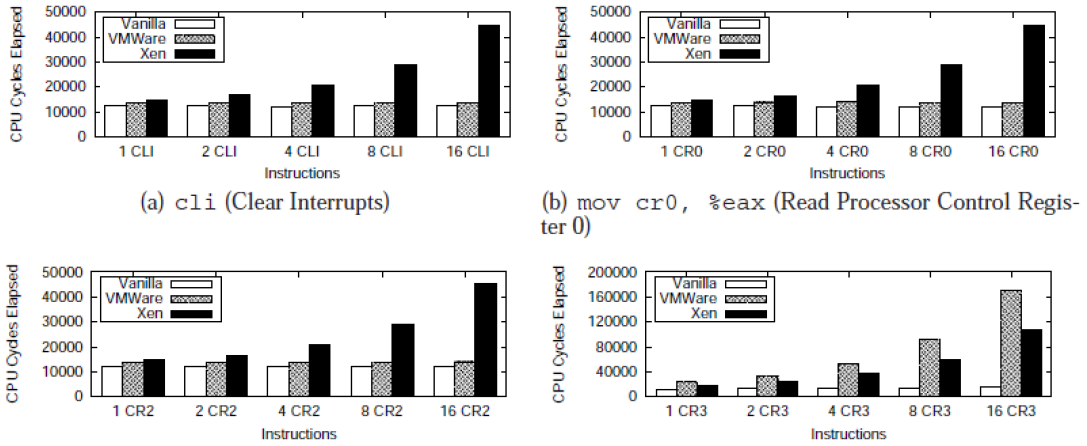
(4) Remote Timing

One of the earliest remote timing detection methods was introduced by a team from Carnegie Mellon in 2008, referred to as "fuzzy benchmarking" [35]. The fuzzy benchmark method exploits timing dependencies of virtual machines to reveal their measurable overhead [35].

The fuzzy benchmark method relies on the assumptions that an external, remote verifier has root access to the target machine and is capable of installing and running benchmarking software within that machine, that the remote verifier has a timing mechanism capable of observing the execution time of the software on the target machine, and that the remote verifier has a means of obtaining hardware artifacts from the target machine to perform deductions on the hardware configuration of the target [35]. Although this technique does not require knowledge of the exact hardware configuration, it does need some knowledge of the architecture in order to be successful. The higher the fidelity of the running hardware configuration on the target, the higher the level of success.

Assuming these conditions are met, the benchmarking software will be initiated remotely which proceeds to execute sensitive instruction sequences on the target designed to invoke hypervisor action, if present. These instruction sequences are looped 2^{17} times in order to induce measurable overhead and the remote verifier measures the completion time of this loop on the target [35]. Once the remote verifier collects the performance data from the target, it compares this data to pre-recorded performance data of a NVM with similar hardware architecture of the target. If a hypervisor is present on the target, the data will indicate a disparity between timing performance of the target machine and the NVM.

Figure 11 illustrates observed local execution times for selected sensitive instructions performed on a NVM, referred to as "vanilla" in the figure, a VMware virtualized machine, and a Xen virtualized machine [35].



(a) `cli` (Clear Interrupts)

(b) `mov cr0, %eax` (Read Processor Control Register 0)

(c) `mov %cr2, %eax` (Read Processor Control Register 2)

(d) `mov %cr3, %eax; mov %eax, %cr3` (Read and then write Processor Control Register 3)

Figure 11. Local Execution Times for Selected Sensitive Instructions. Source: [35].

The authors have demonstrated this method to be successful at not only distinguishing between a virtual machine and a NVM, but also distinguishing between different hypervisors as indicated by varying performances of CR0 and CR3 read/writes, Figure 11(b) and Figure 11(d), respectively. While Figure 11 simply illustrates the differences in local execution times, the fuzzy benchmarking method utilizes remote detection, across the Internet, which yields even stronger identifying results when using CR3 read/writes as indicated by Figure 12. The test in Figure 12 indicates results from machines utilizing a Pentium IV processor (P4), an AMD processor with hardware support for virtualization (HVM), and two different hypervisors (VMWare and Xen) compared to a NVM (Vanilla) [35].

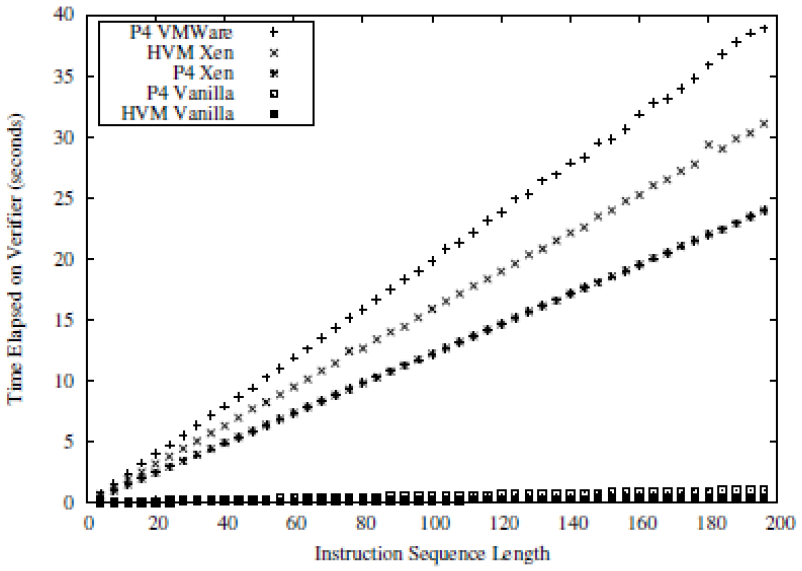


Figure 12. Remote CR3 Read/Write Times. Source: [35].

Figure 12 clearly shows the longer CR3 write times associated with the three hypervisors in comparison to the two NVMs.

b. Counter-Based Detection

Counter-Based Detection (CBD) offers a different approach to detecting hypervisor performance degradations using a counter-based method. While previous methods rely on kernel mode access, CBD methods operate solely in user mode, requiring no elevation of privileges [37]. In addition, CBD uses a counter as measurement rather than a timing source in order to identify the presence of a hypervisor [38]. Figure 13 illustrates the CBD concept.

Figure 13 shows how CBD constructs a race condition between two threads on a multi-threaded CPU. One thread (CPU0) executes a continuous loop of unprivileged instructions such as ADD or NOP, while the second thread (CPU1) executes a continuous loop of CPUID instructions [37]. The ADD instruction is a non-sensitive, non-privileged instruction meaning that the number of times it executes on a NVM will be the same as it would be on a virtualized machine. The ADD execution loop serves as a means of benchmarking the CPUID execution loop. The CPUID instruction is a sensitive, non-privileged instruction which, when executed on a virtualized machine, requires the

hypervisor to perform a VM Exit for every CPUID instruction executed [38]. Every VM Exit results in a delay caused by the hypervisor, which does not occur on a NVM. This induced delay between CPUID instructions is illustrated in Figure 13. Using the ADD execution loop as a baseline, the CPUID execution counter can be calculated for a NVM [38].

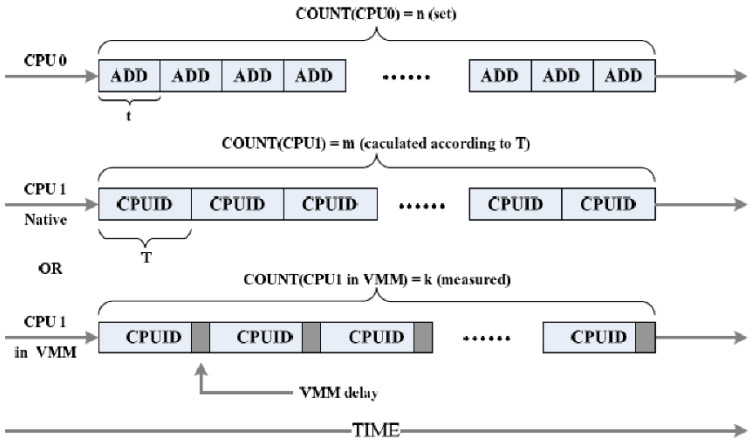


Figure 13. Counter-Based Detection Concept. Source: [38].

When the CBD method is implemented on a machine with a hypervisor, the execution counter becomes skewed due to the induced VM Exit delay, significantly lowering the number of instructions executed and thus revealing the hypervisors presence. Figure 14 demonstrates the variance between the number of CPUID executions on a NVM (clear Windows), a normal hypervisor (normal VMM), and a hypervisor implementing anti-detection methods (anti-time VMM).

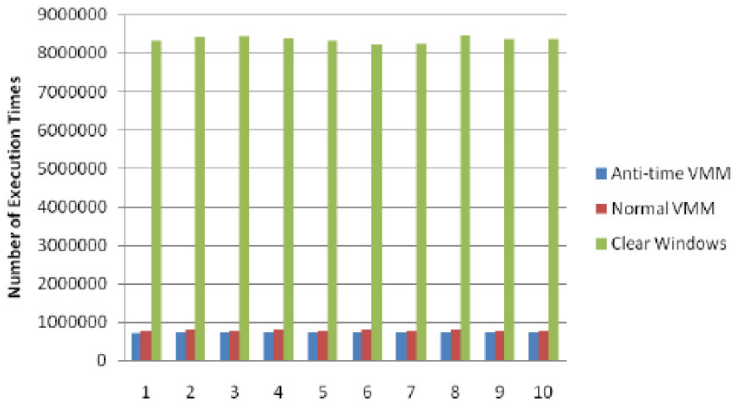


Figure 14. CBD Effectiveness. Source: [38].

As demonstrated in Figure 14, CBD is a very effective method of detecting a hypervisor while affording the flexibility of operating entirely in user space. A NVM achieves much higher counter execution times than that of the two hypervisors tested, making it clearly distinguishable.

c. Graphical

Li et al. introduced a method of VM detection by measuring frame rates and 3D rendering performance [24]. They compared the frame rates between a physical host machine and a virtual machine while running a video game from a browser using WebGL. Li et al. chose to use an application layer approach in order to not only detect virtual machines but also virtual appliances. Virtual appliances are a type of virtual machine designed for a single, specialized purpose that typically only run one application [24]. Figure 15 shows the results of their frame rate testing.

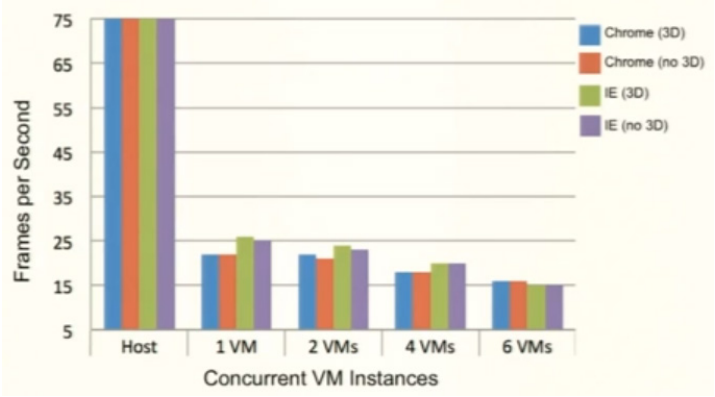


Figure 15. Frame Rate Performance Comparison. Source: [24].

Figure 15 illustrates the higher frame rate performance by the NVM, labeled as "host" in the figure. Decreasing frame rate is observed in virtual machines and continues to decline as the number of concurrent VMs on the hypervisor increases. Therefore, it is possible to differentiate a NVM from a virtual machine using their web script.

Another method of detection introduced by Li et al. measured the performance of 3D rendering. They implemented a web script that measured the number of 3D boxes that could be rendered at a fixed frame rate in a NVM compared to rendering performance in virtual machines [24]. Figure 16 illustrates the observations from this experiment.

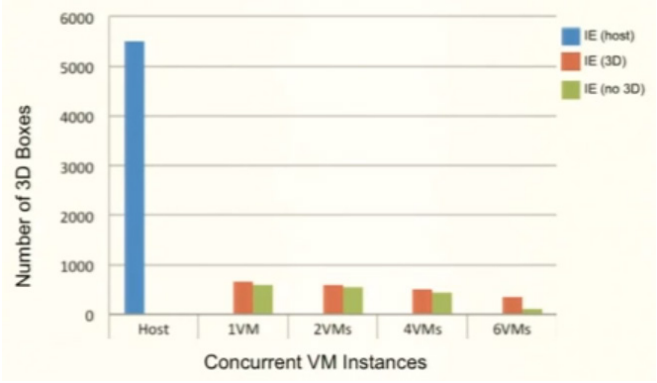


Figure 16. 3D Rendering Performance Comparison. Source: [24].

A clear performance difference can be observed whereby the NVM had a much higher 3D rendering capability as compared to virtual machines. In addition, the 3D rendering

performance can be seen to decline as the number of concurrent VMs increases. This research suggests graphical performance can be used to accurately detect virtual machines.

4. Security

The security domain seeks to identify the differences between a NVM and a virtual machine from a security standpoint. Virtualization inherently introduces security threats and vulnerabilities that do not exist on NVMs. Identifying these threats and vulnerabilities is crucial from the standpoint of HFV. A High-Fidelity Hypervisor (HFH) must be aware of these security characteristics in order to implement mitigation techniques suitable for a HFV environment. However, the task of identifying and categorizing virtualization security concerns is not a trivial process and much research has already been conducted in this field. Many researchers have offered different approaches to identifying and categorizing virtualization security threats and vulnerabilities.

The most recent and exhaustive work was conducted by Patil and Modi in 2019 [39]. We assessed their work to be the most comprehensive study in the field of virtualization security, and therefore chose to implement their taxonomy in our HFV security domain. Figure 17 illustrates their taxonomy of virtualization-related security attacks.

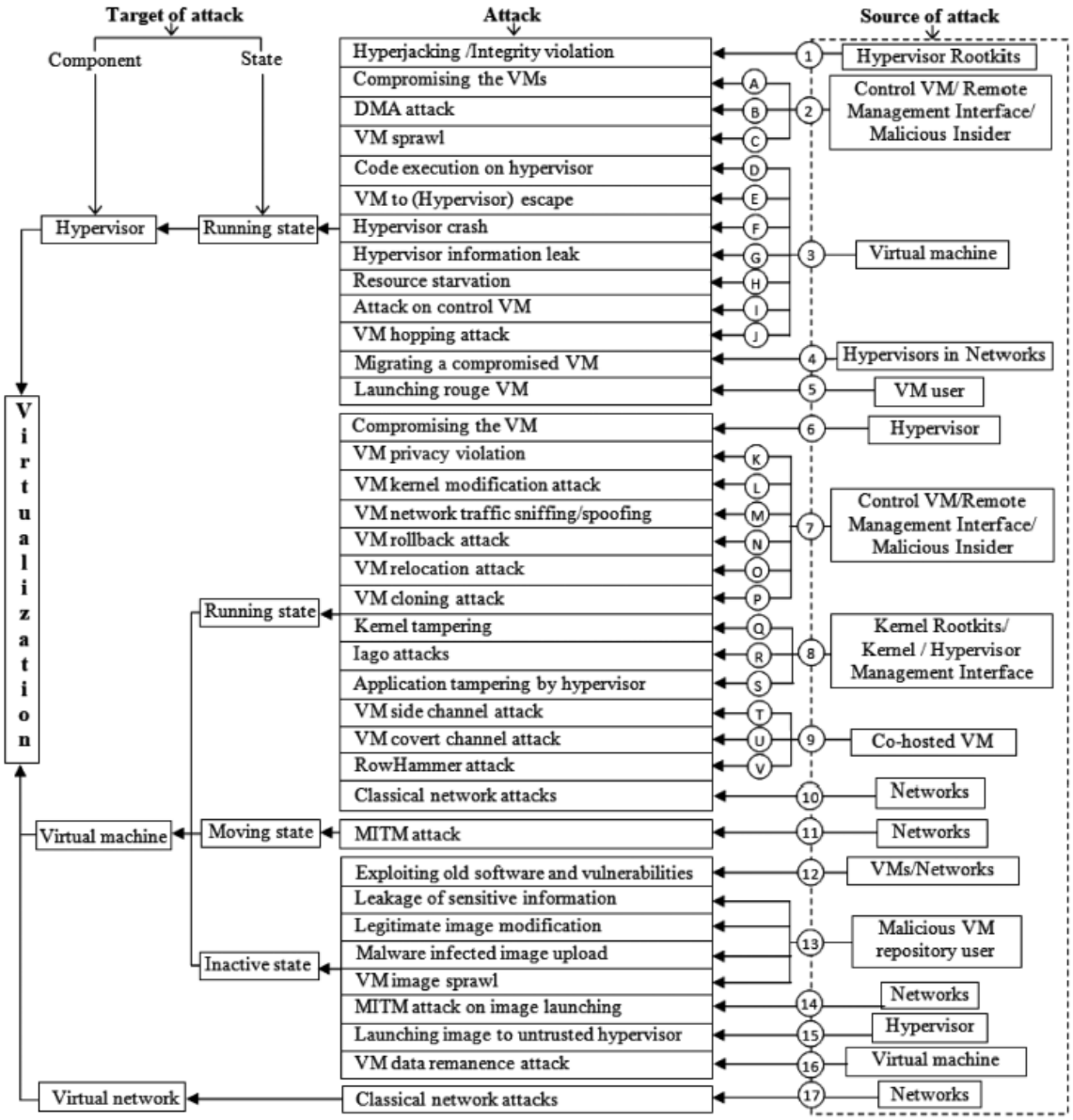


Figure 17. Taxonomy of Virtualization-Related Security Attacks.
Source: [39].

Patil and Modi identified three main categories of virtualization attack surfaces: hypervisors, virtual machines, and virtual networks [39]. Virtual networks are beyond the scope of this thesis and will not be discussed in this chapter. Utilizing the taxonomy presented by Patil and Modi, we classify two sub-domains for HFV security characteristics: hypervisors and virtual machines.

a. Hypervisor Security

Hypervisors introduce a large attack surface due to their inherent role as the foundational layer of abstraction necessary to create a virtualization environment. We assessed hypervisor security by examining the vulnerabilities, threats, and attacks associated with hypervisors in accordance with the model set forth by Patil and Modi [39].

(1) Vulnerabilities

Vulnerabilities in the hypervisor vary by hypervisor vendor. Many of these vulnerabilities exist due to bugs or poor design by the hypervisor vendor, inadequate control over privileged management interfaces, and a lack of control over VM resource allocation, to name a few [39]. The Xen hypervisor alone has over 90 vulnerabilities documented in the National Vulnerability Database within the last two years. Most of these vulnerabilities stem from hypervisor design flaws such as out-of-bounds read/write access, infinite loops, and NULL pointer dereferences [39].

(2) Threats

The threats associated with hypervisors commonly include compromised management interfaces, uncontrolled VM growth, and unauthorized access to hypervisor resources [39]. A hypervisor threat model developed by Patil and Modi is illustrated in Figure 18. A compromised hypervisor can have cascading effects in a virtualized system, as demonstrated by Figure 18. Flawed hypervisor implementation introduces myriad security issues, which can affect not only the hypervisor itself, but also VMs under its purview to include the virtual interfaces of each VM.

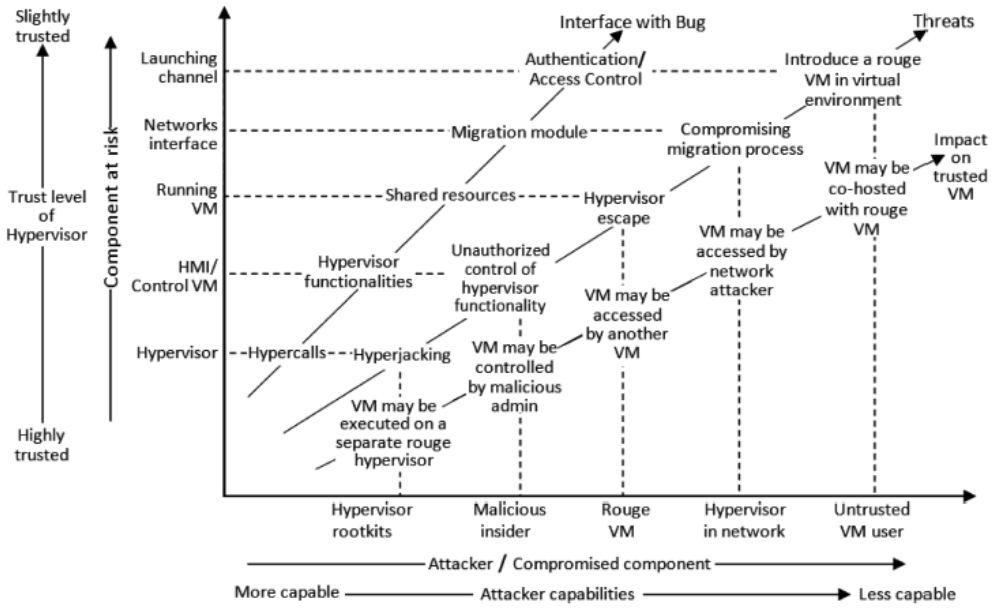


Figure 18. Hypervisor Threat Model. Source: [39].

(3) Attacks

There are many examples of hypervisor attacks as discussed in great detail by Patil and Modi [39]. We will discuss two of these attacks as examples of the types of characteristics relative to the HFV security domain.

Hyperjacking is a term used to describe a rootkit that attempts to gain control of a hypervisor [39]. The rootkit will then deploy a VMBR between host hardware and the OS in order to establish interposition beneath the previously installed hypervisor. This allows the VMBR to view and control all interactions between the hypervisor and host hardware, essentially gaining control over the host and guest machine [39]. Hyperjacking is an attack unique to virtualized environments and represents a difference in security characteristics between virtualized and NVMs.

Many attacks on the hypervisor may come from the VMs it hosts. Most hypervisors assume their own VMs to be secure, but they can in fact be compromised by an insider threat or other malicious software running on the VM OS [39]. The VM can potentially perform malicious actions on the hypervisor such as escaping the VM due to hypervisor isolation failure, executing code on the hypervisor or leaking hypervisor information via

inadequately controlled shared resources, and effectively performing a denial of service attack on the hypervisor by consuming all of the host resources [39].

b. Virtual Machine Security

Virtual machines represent the second inherent security challenge in virtualization technology. VMs are the "edge" of virtualization and therefore are exposed to many different interfaces that could be used as attack vectors such as drivers, malicious users, and remote access interfaces [39].

(1) Vulnerabilities

Patil and Modi suggest three states from which VM vulnerabilities are present: the running state, the migration state, and the inactive state. The migration and inactive states are beyond the scope of the HFV security domain and will not be addressed in this thesis. HFV solely seeks to capture a running state that more holistically represents a NVM. Implementing security measures while the HFV machine is powered down are considered implied good security practices that are not directly related to HFV. Any security concerns in these states are not to be discounted, but their implementation is orthogonal to the implementation of HFV.

Vulnerabilities associated with VMs in the running state may include inadequate VM and shared resource isolation, inadequate control over VM rollbacks and management interfaces, and default VM creation states [39].

(2) Threats

Common VM threats include illegal access from the hypervisor management interface, rogue VMs, and VM specific rootkits [39]. Figure 19 illustrates the VM threat model as presented by Patil and Modi. VMs can be compromised by access to the control VM, infected VM images, and malicious co-hosted VMs [39].

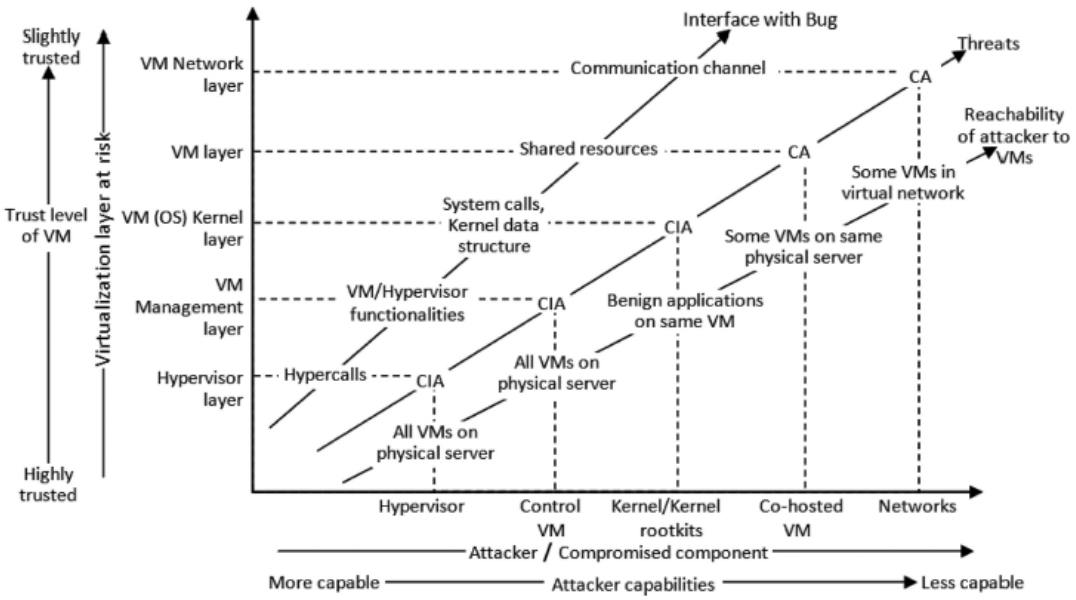


Figure 19. Virtual Machine Threat Model. Source: [39].

(3) Attacks

Patil and Modi provide a comprehensive review of VM attacks; however, we will only discuss two attack surfaces as examples of the VM sub-domain.

Row hammer is a known exploit in certain Dynamic Random-Access Memory modules that can result in inadvertent fluctuations of bits in memory. These fluctuations are a result of the high memory cell density of Dynamic RAM which leads to electrical interference between neighboring cells during memory access [40]. Several researchers from Ohio State University demonstrated methods to break virtualization memory isolation to access host physical memory by utilizing the row hammer exploit [40]. Although the row hammer exploit by itself is not unique to virtualization, when combined with other methods it can be used to violate the safety property of virtual machines and represents a HFV security characteristic.

VM isolation failures may introduce the presence of side or covert channels being established between co-hosted VMs [39]. A side-channel can be used to leak information from one VM to another by using shared hardware caches. Covert channels can leak information also using shared hardware caches, shared memory, and achieve high

bandwidth extraction using memory buses [39]. Side-channel and covert channel attacks represent a security risk that exists when multiple VMs are co-hosted on the same hypervisor, and must be considered when operating multiple HFV machines on a single platform.

5. Functionality

The functionality domain includes the characteristics that differentiate a virtual machine from a NVM from the user perspective. The functionality domain identifies the functional limitations virtual machines introduce that are not present in NVMs. Functional characteristics are represented by any capability a user has on a NVM that they do not have on a virtual machine.

The biggest limitation on user functionality inside a VM compared to functionality inside a NVM, is typically related to any interactions with virtual hardware, or physical hardware from the perspective of the user. Inherent to virtualization is safety, as set forth by Popek and Goldberg in their requirements for a virtual machine. Safety is enforced in virtual machines by creating isolation from host hardware through interposition of a hypervisor. This isolation prevents user interactions with hardware that would typically be possible in a NVM. For instance, the program "HWinfo" allows users to obtain hardware sensor information for their machine by installing a kernel mode driver. On a NVM this software can be used to obtain a multitude of hardware sensor data to include CPU temperature and fan speed, Graphics Processing Unit (GPU) temperature and fan speed, battery level, voltage, and capacity, RAM temperature, chipset temperature, and hard drive temperature. However, hardware sensor data cannot be obtained in a virtual machine for two reasons: the safety requirement isolates the VM from physical hardware, and virtual hardware emulated by the hypervisor is not emulated to a high enough degree of fidelity to support retrieving and interacting with physical sensor data. The functionality domain seeks to create high-fidelity virtual hardware that can be incorporated into the HFV model.

In this section, we have introduced the functionality domain. Greater details on how this domain could be beneficial for HFV as well as a methodology for its implementation are described in Chapter 4.

C. SUMMARY

This chapter proposed five domains for HFV characteristics that could potentially be implemented to create a HFVM. These domains represent a unique approach to HFV and bring unique challenges in order to be successfully implemented. Many hypervisor developers today have focused on providing end users with efficient, scalable and portable hypervisors in order to facilitate enterprise needs for virtualization. HFV presents a new approach to hypervisor design by implementing one or all domains presented in this chapter. The next chapter analyzes each of the five domains for their suitability and feasibility for HFV implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. HFV CHARACTERISTICS ASSESSMENT

A. INTRODUCTION

As demonstrated by the survey of characteristics in Chapter 3, HFV is a broad concept that offers many different levels of design and opportunities for implementation. Since the hypervisor is the foundational layer of virtualization technology, it is perhaps the most logical method to implement HFV. A HFH must be designed to incorporate, or in some cases mitigate, as many HFV characteristics as possible based on the particular purpose of the specific HFV implementation. As such, each HFV domain must be closely considered to analyze the benefits, challenges, and potential methodologies that would be unique to its implementation in a HFH.

B. HFV DOMAIN ASSESSMENT

Each domain is examined below using criteria with respect to HFV: the benefits of incorporating their characteristics in HFV, special challenges and considerations associated with each domain, and potential methodologies for HFV domains that are not currently being implemented or considered in virtualization technology today. This assessment was conducted based on the goal of answering how a VM can be created to more holistically emulate a NVM.

1. Artifacts

Virtualization artifacts can be thought of as artifacts visible on the surface of the HFV architecture and represent the appearance of the VM from the perspective of its users. Altering the appearance of the VM provides the initial veil for HFV. If VM artifacts are not taken into consideration, the VM will be immediately recognizable despite successful HFV implementations in all other domains. Therefore, mitigating virtualization artifacts represents the first step toward achieving the goal of a comprehensive HFV implementation.

a. HFV Processes, File System, and Registry Considerations

Commercial hypervisor technology today is designed based on principles of efficiency, scalability, and portability, but not high-fidelity. These design principles lead to artifact creation that could be easily avoided in a hypervisor designed around the concept of HFV. This concept has already been introduced in several HVM rootkits, namely Blue Pill and Vitriol [28], [41]. Artifact mitigation in these cases was established by creating an ultra-thin hypervisor, requiring no Basic Input Output System (BIOS), boot sector, or persistent storage modifications [21]. A HFH would certainly be more robust in design than an "ultra-thin" hypervisor and would therefore require a more tailored design approach; however, removing many virtualization artifacts has already been demonstrated. This capability was demonstrated by Tamas Lengyel at Hacktivity 2016 using a program called Drakvuf [42]. Drakvuf is a hypervisor-based malware analysis tool, designed specifically for Xen, to monitor VM security from a separate security stack outside the guest VM [43]. Tamas demonstrated that many Xen hypervisor artifacts could be manipulated before being returned to the VM to avoid detection by specialized VM detection software such as Paranoid Fish [42].

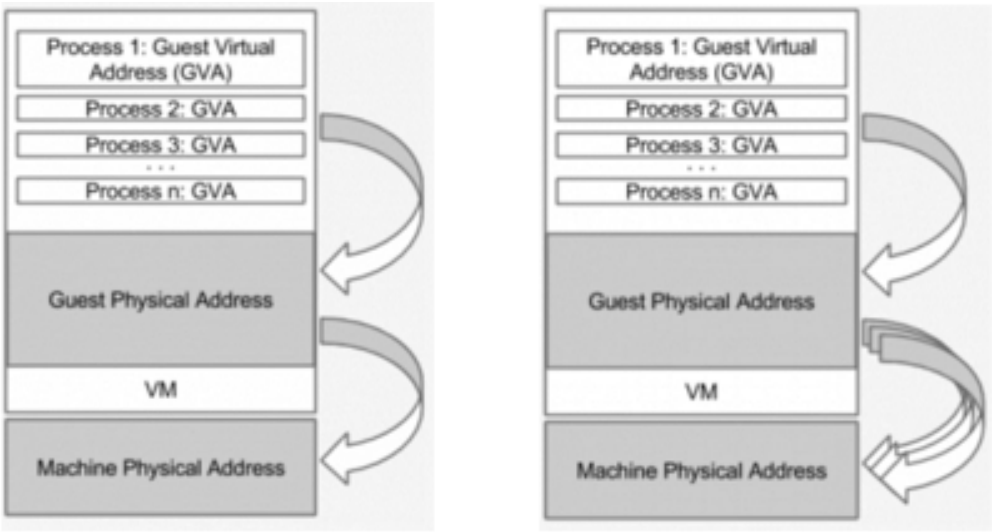
b. HFV Resource Allocation Considerations

Drakvuf was also demonstrated to defeat Paranoid Fish detection methods based on resource allocation [42]. Paranoid Fish attempts to detect VMs by checking hard drive space, memory allocation, and number of CPUs running on the machine [42]. Lengyel demonstrated that Drakvuf could be configured to defeat these detection mechanisms using Logical Volume Manger and Xen memory sharing Copy-on-Write (CoW) support. CoW is a virtual resource allocation method whereby the full allocation is dynamically performed, as the system requires the resource. Utilizing CoW, 250Gb can be allocated to a VM, but the VM will not use the full 250Gb until it needs the space. Using CoW, both the VM hard drive and memory can be allocated in a manner consistent with NVM resource allocation while still preserving host resources.

A HFH represents a specialized virtual environment and could therefore also defeat resource allocation detection methods by simply allocating resources similar to a NVM. A

HFH should be installed on a robust physical machine with ample resources to allow for adequate VM resource allocation or should incorporate CoW allocation methods to present the façade of NVM resource allocation.

Defeating methods that are designed to detect the number of operational CPUs is a more difficult task; however, Lengyel also demonstrated this capability with Drakvuf [42]. Lengyel was able to defeat Paranoid Fish CPU detection methods by eliminating race conditions between virtual CPUs, allowing multiple virtual CPUs to be run safely in a virtualized environment. Drakvuf achieved this capability by utilizing Intel's virtualization technology, which supports multiple Extended Page Tables (EPT) for virtualized environments, and Xen's altP2M technology, which supports up to ten EPTs to be assigned to each virtual CPU [42], [44]. AltP2M is a clever method of changing EPT pointers by allowing the hypervisor to change the guest-physical to host-physical memory mapping on the fly [42]. These pointers can be independently adjusted for each virtual CPU and allow the hypervisor to manipulate what the guest can see in memory by changing the EPT to point to different areas in host-physical memory. This process is illustrated in Figure 20.



At left, single EPT memory mapping is illustrated whereas at right, altP2M is illustrated demonstrating the capability to map guest-physical addresses to multiple machine-physical addresses, which represent host-physical memory.

Figure 20. EPT Memory Mapping. Source: [42].

A HFH would benefit from implementing altP2M technology, or something similar, that takes advantage of Intel's hardware support for multiple EPTs. Similar to Drakvuf, a HFH could use multiple EPTs to ensure other hypervisor artifacts remain hidden or even potentially mask unique hypervisor behavior to avoid detection.

c. HFV Memory Artifact Considerations

A HFH must also account for artifact detection through memory pointers. The SIDT instruction used to detect the presence of a hypervisor, as discussed in the Chapter III, are all non-privileged instructions that will not cause a trap to the hypervisor, which is partly why they are so effective. To counteract this, a HFH would need to detect these instructions being executed and emulate a response back to the VM that would align with the appropriate NVM response.

2. Behavior

A behavioral difference that can be observed between an NVM and VM can be used to distinguish between the two machine types. HFV seeks to create a VM that more holistically captures NVM behavior, from the perspective of the guest VM. Therefore, it is crucial that all known HFV behavioral characteristics be considered when designing a HFH. A hypervisor that behaves like an NVM, or emulates NVM behavior more holistically, represents a significant step toward a comprehensive HFV implementation. A HFH that can successfully account for behavioral differences would provide a significant benefit in virtualization technology. Emulating NVM behavior at a high level of fidelity would make it significantly harder to detect the presence of the hypervisor. Behavior detection avoidance mechanisms would allow unique opportunities to develop and test software designed to take advantage of key NVM behavioral characteristics on a VM. This technology would allow high-fidelity, flexible, and secure testing environments that could be duplicated, adapted, or reset on the fly.

A HFH would require stringent limitations from that of a traditional commercial hypervisor in order to be effective in mitigating identifying behaviors of VMs. Perhaps the most limiting mitigation would be to require the hypervisor to host only a single VM at a time. This requirement could be relaxed depending on the goals of the specific HFV

implementation, however, a hypervisor with co-hosted VMs could lead to detection via behavior methods as demonstrated by the team from Tokyo University of Technology [30], [32]. Noorifiza et al. [32] postulated that identifying behavior was observed due to the hypervisor's simultaneous management of co-hosted VMs. Running a specialized, single VM on a HFH could help mitigate this behavior by eliminating the need for the hypervisor to multitask resources across multiple VMs.

In addition, other options to specifically mitigate timestamping behavior include modifications in hypervisor design that enable timestamp manipulation. Timestamp manipulation at the hypervisor level could allow VM timestamp behavior to more closely align with NVM timestamp behavior by ensuring that all timestamps are encoded in a manner consistent with NVM timestamps. This added interposition function by the hypervisor would inadvertently lead to a potentially measurable time delay however, which could enable other hypervisor detection methods.

A HFH would also not be able to utilize a guest-to-host communication channel, or any other hypervisor-specific functionality designed to improve end-user quality of service. Since the guest-to-host communication channel is easily detectable, conventional communication channels would need to be used for the host to communicate with the guest.

Unfortunately, a HFH cannot address the unique problems presented by the behavior domain independently. As previously discussed in Chapter III, CPU behavior is also a significant indicator of the presence of virtualization. In order to fully implement the HFV behavior domain, adjustments in hardware support for virtualization must be made. These modifications are not trivial however and would require full redesign of CPU microarchitecture to satisfy HFV goals. From the behavior perspective, while modifications to CPU caches such as the TLB and RSB can be observed or detected during key virtualization events such as VM Exit, successful virtualization detection methods will persist. CPU redesign for the purposes of HFV is not practical in the near future, but as the use of virtualization continues to expand in cyber space, processor design could evolve to accommodate HFV principles.

3. Performance

The performance domain is perhaps the most relevant domain not only to HFV, but virtualization technology in general. The performance domain is unique because it is the only HFV domain currently addressed by design considerations of commercial hypervisors on the market today. Performance is a crucial consideration in hypervisor design, and manufacturers must ensure they obey the efficiency property set forth by Popek and Goldberg to create a hypervisor that executes in as close to real-time NVM execution as possible. Although hypervisor vendors have developed efficient methods of implementing performance improvements, virtualization overhead is inherent to the concept of interposition through an added layer of abstraction between the hardware and VM, and cannot be fully eliminated. A comprehensive HFV implementation would seek to further mitigate the detection of performance differences between NVMs and VMs.

a. HFV Timing Considerations

A HFH could be extended to defeat local timing performance detection mechanisms by modifying the timing value returned back to the guest VM, making it appear as if there is no timing delay [19], [26]. Any timing measurement that utilizes local VM timing sources is susceptible to hypervisor introspection and can be influenced by the hypervisor. A HFH must incorporate mechanisms to defeat local timing detection methods to align with the concept of HFV.

Remote detection methods are much more difficult for a HFH to overcome. If the observer utilizes an accurate remote timing source, the hypervisor cannot influence timing measurements directly and other mitigation methods must be developed. Most remote timing detection techniques require many iterations, often thousands, of specific instructions on the guest VM to induce a measurable timing delay from a remote source [26]. Blue Pill introduced a method to defeat remote timing detection mechanisms called Blue Chicken, whereby the hypervisor detects these repetitive iterations being executed as abnormal behavior [41]. Once Blue Pill detects that a remote timing test is being initiated on the VM, it uninstalls itself from the machine temporarily, setting a timer in the kernel for reinstallation. This process allows the remaining timing test to be executed directly on

physical hardware, without hypervisor interposition, resulting in the remote detector perceiving what appears to be NVM timing performance. After the timer has elapsed, Blue Pill reinstalls itself on the machine and restores hypervisor interposition.

Blue Chicken is a clever method of mitigating remote timing detection methods; however, it is not feasible for implementation in HFV. Blue Pill is a VMBR that consists of an ultra-thin hypervisor designed solely for the purposes of interposition; as such, it does not obey the rule of safety as set forth by Popek and Goldberg because isolation is not a goal of VMBRs [41]. Blue Pill can be easily loaded and unloaded on a machine due to its minimal footprint and requires no modification of the BIOS, boot sector, or system files [41]. A comprehensive HFH will require a robust, persistent installation to accommodate all HFV features. Loading and unloading a HFH on the fly will not be practical, therefore other methods of mitigating remote timing methods must be developed.

A HFH could use the same detection techniques as Blue Pill to determine when a remote timing measurement is being executed in its guest VM. Rather than uninstalling upon detection, the hypervisor could halt the timing code execution and emulate an appropriate timing response based on the number of instruction iterations performed by the remote detector. This process would require prior knowledge of the code's signature to enable detection and development of an appropriate and dynamic emulated timing response to be sent to the guest VM. As a result, this mitigation effort would be highly specialized and incapable of providing a solution for all remote timing techniques. Therefore, a HFH would be still be susceptible to remote timing detection methods.

b. HFV CBD Considerations

Counter-Based Detection (CBD) methods would also prove difficult to overcome in any HFV implementation. Since CBD requires no timing source and does not require kernel level privileges to execute, a HFH can do very little to mitigate this technique. As hardware support for virtualization allows unconditionally intercepted instructions—instructions that always cause a VM exit when executed—to be executed from user mode, the ability to mitigate performance detection methods with software is near impossible. In the case of Intel, the CPUID instruction is an unconditionally intercepted instruction that

can be run in any privilege level [15]. While the VM is executing on an Intel CPU, a CPUID instruction will always cause a VM exit. This exit is a safety feature designed to ensure the guest VM does not gain direct access to the host CPU; however, it also enables an easy method to detect performance differences in VMs such as the method implemented in CBD. The CPUID instruction presents a bit of a paradox with respect to HFV. In an HFV model, the CPUID should still be capable of executing at any privilege level in order to support instruction serialization and instruction set fidelity. However, HFV would also seek to eliminate the CPUID instruction from causing an unconditional interception by the hypervisor to mitigate performance-measuring methods but this would violate the safety requirement set forth by Popek and Goldberg and give the VM direct access to the CPU. For these reasons, the performance domain poses the greatest challenges for HFV and may not be capable of being fully implemented without significant changes to hardware support for virtualization.

c. HFV Graphical Considerations

Graphical performance monitoring represents another challenge in HFV. 3D rendering performance in a VM is a blatant indicator of virtualization and must be addressed by HFV. The research conducted by Li et al. [24] was specifically designed to detect virtual appliances, a subset of virtual machines. Their research was primarily focused on detecting performance differences of virtual GPUs that are emulated in software. However, there are some GPU virtualization technologies that make graphical performance detection methods more difficult, such as GPU passthrough or directed I/O. Li et al. addressed this technology in their research, but their experiment was designed to detect virtual appliances, which would not normally use GPU passthrough technology. Intel released the capability to directly connect I/O devices to VMs with the release of Intel VT-d, which stands for Directed I/O, in 2007 [44]. Intel VT-d enables GPU passthrough functionality which allows VMs to directly connect to the GPU without interaction from the hypervisor, reducing virtualization overhead and greatly improving graphical performance [45]. Although graphical performance can be significantly improved by using direct passthrough methods, some researchers have still identified measurable differences when compared to NVM performance. Shea and Liu discovered that GPU performance

dropped from 85 FPS on a NVM, to 51 FPS on a VM when using direct GPU passthrough [46]. Although this is a significant performance increase over the graphical performance noted by Li et al., it still represents a measurable performance difference that could be disadvantageous for HFV.

A HFH would most certainly need to take advantage of GPU passthrough technology, along with other directed I/O devices, to improve performance to near native speeds. Although GPU passthrough may not provide a comprehensive solution, it would provide worthwhile mitigation to graphical performance detection methods.

4. Security

HFV represents a subset of virtualization technology and therefore is inherently vulnerable to security concerns unique to the virtualization domain. Traditional virtualization security must only mitigate or defend against vulnerabilities, threats, and attacks specific to virtualized environments. HFV security must go one step further, however, by performing mitigation and defense without being detected from within the HFVM. Perhaps the best way to enforce HFV security goals is to isolate the security stack outside the HFVM. Drakvuf, mentioned previously in this chapter, offers a virtualization security solution that does just that and would perfectly complement HFV objectives in the security domain.

Drakvuf is an extension of the Xen hypervisor which manages VM security from an external security stack that is isolated from the rest of the hypervisor [43]. It provides stealthy malware analysis through a nearly undetectable footprint from inside the VM [47]. Drakvuf uses a process hijacking procedure from the hypervisor to execute arbitrary code inside the VM [47]. This technique allows the hypervisor to conduct thorough malware analysis on guest VMs without their knowledge. Coupled with the stealthy nature of Drakvuf's profile, it offers a complementary security solution that could be integrated into a HFH. A HFH with a Drakvuf extension would allow the hypervisor to mitigate artifact and behavioral characteristics, as mentioned previously in this chapter, in addition to providing security oversight for HFVMs. The additional interposition of the Drakvuf

extension, incorporated with additional HFV goals such as NVM security emulation, would offer a tailored, robust security solution for HFV.

5. Functionality

The functionality domain is a critical element of HFV from the guest perspective. A HFH must be able to accurately emulate the user functionality of a NVM within a HFVM. The user must be unable to observe a measurable functional difference in the HFVM for it to be successful at presenting holistic HFV.

HFV must emulate virtual hardware with a high degree of fidelity to achieve the goals set forth in the functionality domain. Many hypervisor vendors do not offer high fidelity virtual hardware simply because there is no need for it in today's virtualization market and writing high-fidelity virtual hardware entirely in software is not a trivial task [48]. However, as virtualization becomes more ubiquitous in cyber space and VMs become more specialized, the need for high-fidelity virtual hardware could grow. High-fidelity virtual hardware would provide a HFVM that more holistically represents a NVM, and would contribute to other HFV domains by mitigating detection methods based on virtual hardware artifacts or virtual hardware behavior. The ability to create high-fidelity virtual hardware that responds to induced failure conditions that cascade into the HFVM could greatly contribute to more realistic cyber operations. For instance, a virtual CPU fan malfunction could cause an overheating condition that automatically shuts down the HFVM.

a. HFV Methodology for Cyber-Physical Systems

As discussed in Chapter II Zhang et al. introduced a methodology for creating a holistic execution environment for a cyber-physical system by synchronizing a VM with a Physical Component Emulator (PCE) in MATLAB [18]. By adapting their methodology to our definition of HFV, it is possible to create high-fidelity virtual hardware that could be utilized for the functionality domain. As they point out, to holistically capture the relationship between cyber and physical components, a fully closed-loop, synchronized execution environment must be developed that can accurately capture the interdependent behavior between the two components [18]. Synchronization between cyber and physical

components is crucial to achieving high-fidelity [18]. Their experiment utilized differential equations to create a PCE in MATLAB and synchronized it with a VM using a specialized synchronization protocol they developed to model the physical components represented by a time continuum and the cyber components represented by discrete, step by step events [18]. This protocol, represented in Figure 21, is a continuous feedback loop between the state of the cyber component and the calculation of the differential equation in the PCE.

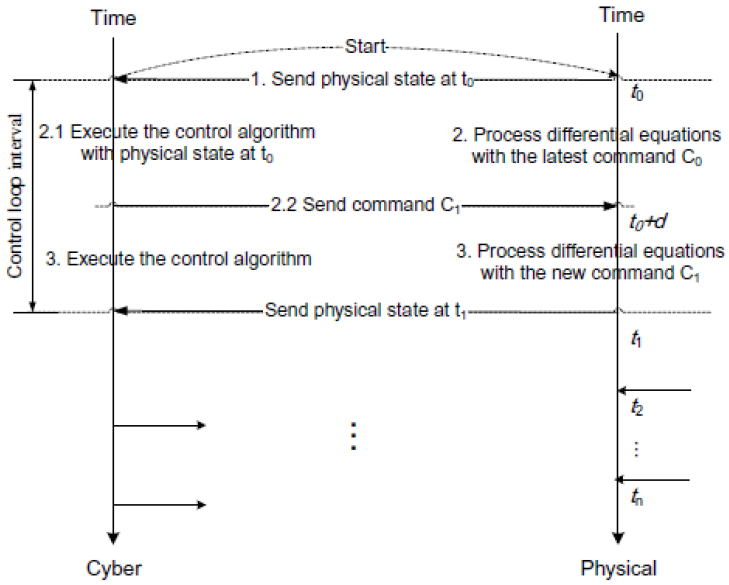


Figure 21. Synchronization Protocol between Cyber Component and Physical Component. Source: [18].

The holistic execution environment implementing the synchronization protocol is illustrated in Figure 22. This virtual execution environment is created by providing an interface between a QEMU virtual machine running controller software and MATLAB, which is emulating state data of physical components. QEMU and MATLAB are linked together by the synchronization protocol (represented by the bottom portion of Figure 22) to create a single logical execution environment (represented by the upper portion of Figure 22). This logical execution environment allows the interconnected programs to holistically emulate a cyber physical system.

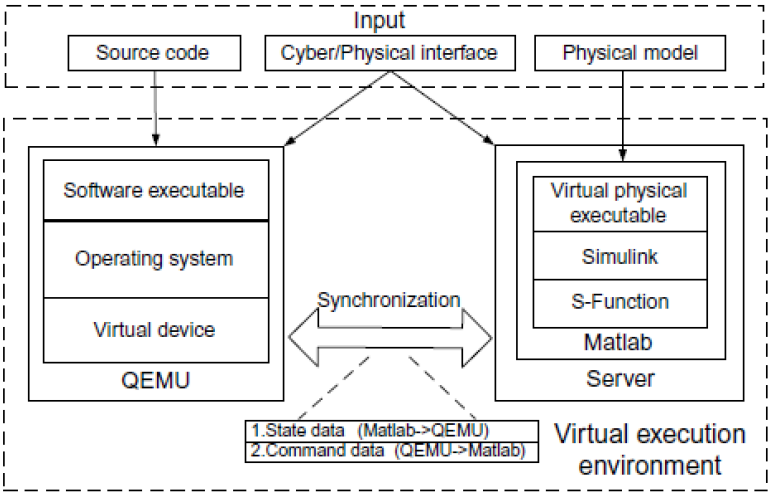


Figure 22. Zhang et al. Holistic Execution Environment. Source: [18].

b. HFV Methodology for Desktop Virtualization

By adapting the work of Zhang et al., a HFV methodology can be established to create high-fidelity virtual hardware for desktop virtualization. Figure 23 illustrates the high-fidelity virtual hardware holistic execution environment.

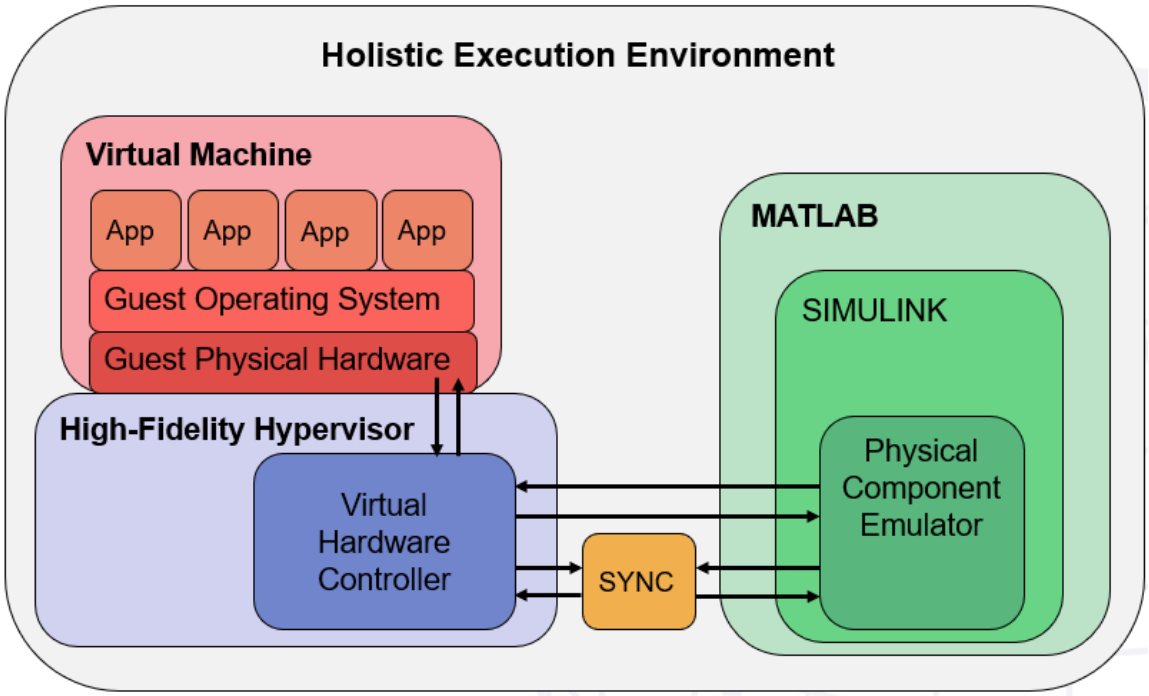


Figure 23. Virtual Hardware Holistic Execution Environment.

The cyber component would be represented by a HFH with a VM installed. The HFH would have a Virtual Hardware Controller (VHC) module that interfaces with the PCE and the synchronization module. The PCE would consist of a single, physical hardware characteristic mapped by a differential equation in MATLAB. Utilizing the synchronization protocol introduced by Zhang et al., the VHC could be synchronized to the PCE to allow physical state data to be calculated based on discrete events that occur in the VM. High-fidelity virtual hardware could be created by feeding emulated physical hardware characteristics from the PCE to the VHC. The VHC could be programmed to induce specific behavior on the guest VM when it receives data from the PCE. For example, if the PCE was modelling CPU temperature, the VHC could initiate an overheating condition inside the guest VM when it received high temperatures from the PCE. In contrast, the VHC could also influence the PCE by sending specific state information based on VM performance that would result in a rise or drop in the modelled CPU temperature. The VHC could also feed emulated physical characteristics of the virtual hardware to the

guest VM to create a high-fidelity virtual environment from the user perspective. For example, providing CPU temperature information to the user inside the guest VM.

The creation of high-fidelity virtual hardware using this method would comprise a major contribution to the concept of HFV; however, implementation would not be trivial. Modelling the physical behavior of hardware characteristics using differential equations would likely serve as the biggest challenge in a high-fidelity virtual hardware implementation using this methodology. An initial proof of concept might be implemented using an existing, open-source hypervisor, such as Xen, rather than developing a new hypervisor. Despite the open-source nature of Xen, developing a VHC module that interfaces with the hypervisor in a manner conducive to creating high-fidelity virtual hardware would also provide a challenge. However, with the development of other third-party Xen hypervisor extensions such as Drakvuf, this task is certainly achievable.

C. SUMMARY

This chapter provided an analysis of the potential integration of each HFV characteristic domain into the development of a HFH. It discussed the unique challenges that each domain presents to the HFV concept and offered potential solutions to these challenges. Some HFV domain characteristics have already been developed and integrated into current hypervisors and could be incorporated into a HFH. Other HFV domain characteristics are relatively new concepts that have not yet been implemented in hypervisors, as in the case of the functionality domain. We discussed the benefits of a HFV implementation through the functionality domain and proposed an implementation methodology for specific virtual hardware characteristics to create high-fidelity virtual hardware. The next chapter provides a conclusion and discusses opportunities for future work.

V. CONCLUSION AND FUTURE WORK

A. SUMMARY

This thesis was motivated by the opportunity to extend the capabilities of virtual machines to enable the creation of high-fidelity virtual environments for use in cyber operations. We established a working taxonomy of virtualization domains to appropriately scope this thesis, and reviewed the current state of virtualization technology. We conducted research to aggregate and categorize fidelity variance between virtual machines and their physical counterparts, and classified these variances into five domains based on their unique characteristics and potential for application in HFV. Further, we conducted an in-depth analysis on each domain to assess the challenges and practicality of implementation in a HFV environment. Finally, we presented a methodology to emulate physical characteristics of virtual hardware to create a high-fidelity virtual machine.

B. CONCLUSIONS

The conclusions from this thesis are drawn from the research objective, and three research questions.

1. Research Objective

The objective of this research was to identify potential methods of creating high-fidelity virtual environments conducive to specialized cyber operations. Through our analysis in Chapter IV, we assessed five different domains that can be implemented in varying degrees to create high-fidelity virtual environments.

The artifact, behavior, and performance domains can be integrated into a HFH to improve fidelity by increasing stealth and equivalence. These domains can be incorporated in HFV to help create a more holistic representation of a NVM by making it more difficult to detect the presence of virtualization. A hypervisor that more effectively hides its own presence appears more like a NVM to a user in the guest VM.

The security domain could be integrated into a HFH hypervisor to improve fidelity by mitigating the threats and vulnerabilities associated with virtualization. Enforcing NVM

security from an external security stack outside the HFVM would help provide a secure operating environment that would be difficult to detect by HFVM users.

The functionality domain can be implemented in a HFH to improve fidelity by introducing high-fidelity virtual hardware. High-fidelity virtual hardware would offer a level of NVM emulation not present in current virtualization technology and would enable cyber operations that specifically take advantage of hardware behavior and side effects to occur in a virtualized environment.

These five domains may be implemented in a HFV solution individually, or in tandem, to create varying levels of high-fidelity virtual environments based on the specific objectives of tailored cyber operations.

2. Research Questions

This thesis was bounded by three research questions to guide the research process and support the research objective. Final conclusions for this thesis were derived by answering these questions.

a. What specific characteristics not currently implemented in modern virtualization technology could be included to extend a VM to create a high-fidelity virtualized environment?

We conducted research in Chapter III to identify and categorize the differences between VMs and NVMs. The sheer multitude of variance between the two warranted the creation of five domains to present the varying characteristics in an organized manner. We discovered that the variance between VMs and NVMs can be categorically described by the way they look, behave, and perform from the perspective of the user. We also discovered variance associated with vulnerabilities and overall capability available to the user. These categories of variance were organized into five domains to allow further research in their suitability for HFV implementation in Chapter IV.

b. Which of these domains could be used to influence the behavior of a virtual machine if emulated appropriately?

Based on our analysis in Chapter IV, the functionality domain would be best suited as to influence the behavior of a virtual machine in ways that current virtualization technology cannot achieve. While all HFV domains would offer fidelity improvements over NVMs, the functionality domain offers the most potential improvement that could be used to influence the behavior of a virtual machine in a manner consistent with NVM behavior. The artifacts, performance, and security domains would all improve fidelity; however, their integration into HFV would have little effect at creating a platform for influencing VM behavior. The behavior domain represents a potential for influencing VM behavior but provides a very limited avenue to do so. We defined the behavior domain to consist of CPU behavior and hypervisor behavior. CPU behavior is beyond the scope of this thesis and cannot be applied to HFV. The integration of the hypervisor behavior sub-domain in HFV would mainly consist of removing unique hypervisor behavior that can be used as a form of detection, such as guest-to-host communication channels, rather than introducing new behavior. Therefore, the behavior domain is not as well suited as the functionality domain for the purposes of this thesis. If emulated properly, the functionality domain could provide virtual hardware that more holistically emulates physical hardware. This high-fidelity virtual hardware would introduce physical sensor data into virtual environments that could be manipulated to induce specific VM behavior that mirrors NVM behavior under the same circumstances.

c. How could one of these characteristics be emulated at a level of fidelity necessary to influence VM behavior in the same manner it would influence NVM behavior?

In Chapter IV, we presented a methodology for emulating characteristics in the functionality domain to create a high-fidelity virtual environment. We decided to adopt a similar methodology presented by Zhang et al. because creating high-fidelity virtual hardware for HFV is similar to the virtual cyber-physical system they presented in their work. Zhang et al. created a virtual execution environment to connect a virtual controller to an emulated physical component. We hope HFV can accomplish the same goal by

modifying their methodology to connect a virtual machine to emulated hardware characteristics to effectively extend the functionality of a virtual machine. This extended functionality would be provided in the form of high-fidelity virtual hardware created by our methodology.

C. FUTURE RESEARCH

This research was conducted as a preliminary step toward achieving the larger goal of implementing HFV. This thesis served to establish the foundational work necessary to begin creating a high-fidelity virtual environment. Future work should focus on creating a proof of concept for HFV based on the methodology presented in this thesis. Future work can be broken into three separate endeavors: creating a physical component emulator, a virtual hardware controller, and a high-fidelity hypervisor.

1. Physical Component Emulator

The PCE should be developed to accurately emulate a specific physical hardware characteristic, such as CPU temperature. This characteristic could be modelled in MATLAB using a differential equation to model physical characteristic information based on state data. Further research is required to develop a differential equation that models the physical state of a specific characteristic. The PCE should be designed with further HFV integration in mind and must be able to communicate directly with the VHC.

2. Virtual Hardware Controller

The VHC must be designed to communicate with the PCE and the hypervisor directly. This communication channel will allow the VHC to act as a logic controller between virtual and emulated physical components. The VHC must be capable of dynamically inducing behavior in the VM as well as in the PCE to effectively create a closed loop system. Initial development of the VHC could be achieved by extending a pre-existing, open-source hypervisor. The Xen hypervisor offers a flexible integration platform and would serve as a good candidate for use in the development of the VHC. As demonstrated by Drakwuf, custom modules can be created to integrate with the Xen hypervisor.

3. High-Fidelity Hypervisor

The ultimate goal of this research is to create a hypervisor designed around the principles of stealth, fidelity, and equivalence. Ideally, this hypervisor would be designed to integrate all five HFV domains described in Chapter III. A HFH would mitigate virtualization detection techniques while also providing a platform for the creation of high-fidelity virtual hardware. Extensive further research is required to design and create a HFH.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] P. Kampert, "Taxonomy of virtualization technologies," M.S. thesis, Dept. of ICT, TU Delft, Delft, Netherlands, 2010. [Online]. Available: https://repository.tudelft.nl/islandora/object/uuid:c066d4dd-82b4-4bb2-8ad2-eabb8f9aabc6/

[2] J. Ramos, "Security challenges with virtualization," M.S. thesis, Dept. of Comp. Sec., University of Lisbon, Lisbon, Portugal, 2009. [Online]. Available: http://hdl.handle.net/10451/4541

[3] A. Mann, "Virtualization 101: Technologies, Benefits, and Challenges," Ent. Manag. Assoc., Boulder, CO, USA, 2006. [Online]. Available: http://etomicmail.com/files/dedicated_server/Virtualization%20101.pdf

[4] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization*. San Rafael, CA, USA: Morgan & Claypool Publishers, 2017.

[5] G. Heiser, "The role of virtualization in embedded systems," in *Proc. of the 1st work. on IIES*, Glasgow, Scotland, 2008, pp. 11–16.

[6] G. Heiser, "Virtualization for embedded systems," Open Kern. Labs, Chicago, IL, USA, 2007. [Online]. Available: http://mesl.ucsd.edu/gupta/Teaching/cse291-08/Readings/OK_Virtualization_WP.pdf

[7] A. Singh, "An Introduction to Virtualization," Kernel Thread, January 2004. [Online]. Available: http://www.kernelthread.com/publications/virtualization/

[8] E. G. Mallach, "On the relationship between virtual machines and emulators," in *Proc. of the Work. on Virt. Comp. Sys.*, Cambridge, Massachusetts, USA, 1973, pp. 117–126.

[9] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Comm. of the ACM*, vol. 17, no. 7, pp. 412-421, Jul. 1974. [Online]. Available: https://dl.acm.org/citation.cfm?id=361073

[10] VMware Inc., "Virtualization overview," Palo Alto, CA, USA, 2006. [Online]. Available: https://www.vmware.com/pdf/virtualization.pdf

[11] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, "The evolution of an x86 virtual machine monitor," *ACM SIGOPS Oper. Sys. Rev.*, vol. 44, no. 4, pp. 3-18, Dec. 2010. [Online]. Available: https://dl.acm.org/citation.cfm?id=1899930

[12] K. Adams and O. Agesen, “A comparison of software and hardware techniques for x86 virtualization,” in *Proc. of the 12th Intl. Conf. on ASPLOS*, New York, NY, USA, 2006, pp. 2-13. [Online]. Available: https://dl.acm.org/citation.cfm?id=1168860

[13] VMware Inc., “Understanding full virtualization, paravirtualization, and hardware assist,” Palo Alto, CA, USA, WP-028-PRD-01-01, 2011.

[14] P. Barham *et al.*, “Xen and the art of virtualization,” in *Proc. Of the 19th ACM Symp.*, 2003, pp. 164-177. [Online]. Available: https://dl.acm.org/citation.cfm?id=945462

[15] Intel Inc., “Intel 64 and IA-32 architectures software developer’s manual,” Santa Clara, CA, USA, 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm#three-volume

[16] Intel Inc., “Intel virtualization technology specification for the IA-32 Intel architecture,” Santa Clara, CA, USA, Rep. C97063-002, 2005. [Online]. Available: http://dforeman.cs.binghamton.edu/~foreman./550pages/Readings/intel05virtualization.pdf

[17] Genymotion, “Genymotion desktop.” Accessed December 4, 2018. [Online]. Available: https://www.genymotion.com/desktop/

[18] Y. Zhang, F. Xie, Y. Dong, G. Yang, and X. Zhou, “High fidelity virtualization of cyber-physical systems,” *Int. Journ. of Model. Simul. and Sci. Comp.*, vol. 4, no. 2, Jun. 2013. [Online.] doi: 10.1142/S1793962313400059

[19] I. Korkin, “Two challenges of stealthy hypervisors detection: time cheating and data fluctuations,” in *Conf. on Digit. Foren. Secur. and Law*, 2015. [Online.] Available: https://arxiv.org/pdf/1506.04131

[20] S. T. King and P. M. Chen, “SubVirt: implementing malware with virtual machines,” in *2006 IEEE Symp. on Sec. and Priv.*, Berkeley, CA, USA, 2006. [Online.] doi: 10.1109/SP.2006.38

[21] R. C. Fannon, “An analysis of hardware-assisted virtual machine-based rootkits,” M.S. thesis, Dept. of Comp. Sci., NPS, Monterey, CA, USA, 2014. [Online]. Available: http://hdl.handle.net/10945/42621

[22] T. Liston, E. Skoudis, “On the cutting edge: thwarting virtual machine detection,” presented at SANS at Night, 2006. [Online]. Available: https://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf

[23] S. K, “How to check if a Linux system is physical or virtual machine,” OSTECHNIX, May 23, 2018. [Online]. Available: https://www.ostechnix.com/check-linux-system-physical-virtual-machine/

[24] K. Li and X. Li, “Comprehensive virtual appliance detection,” presented at Black Hat 2014, Singapore, Mar. 2014. [Online]. Available: https://www.blackhat.com/docs/asia-14/materials/Li/Asia-14-Li-Comprehensive-Virtual-Appliance-Detection.pdf

[25] T. Anderson and M. Dahlin, *Operating Systems: Principles and Practice*, 2nd ed. Recursive Books, 2014.

[26] H. Fritsch, “Analysis and detection of virtualization-based rootkits,” B.S. thesis, Dept. of Comp. Sci., TUM, Munich, Germany, 2008. [Online]. Available: http://www.mnm-team.org/pub/Fopras/frit08/PDF-Version/frit08.pdf

[27] M. Brengel, M. Backes, and C. Rossow, “Detecting hardware-assisted virtualization,” in *Detec. of Intr. and Mal., and Vuln. Asse.*, 2016. [Online.] Available: https://link.springer.com/content/pdf/10.1007%2F978-3-319-40667-1_11.pdf

[28] T. Ptacek, N. Lawson, and P. Ferrie, “Don’t tell Joanna, the virtualized rootkit is dead,” presented at Black Hat, Las Vegas, NV, USA, Aug. 2007. [Online]. Available: https://archive.org/details/2007_BlackHat_Vegas-V18-Ptacek-Ferrie-Lawson-Dont_Tell_Joanna

[29] G. Maisuradze and C. Rossow, “ret2spec: speculative execution using return stack buffers,” in *Proc. of the 2018 ACM SIGSAC Conf.on CCS*, Toronto, Canada, 2018. [Online.] doi: 10.1145/3243734.3243761

[30] M. Noorafiza, H. Maeda, T. Kinoshita, and R. Uda, “Virtual machine remote detection method using network timestamp in cloud computing,” in *8th Intnl. Conf. for ICITST*, London, United Kingdom, 2013. [Online.] doi: 10.1109/ICITST.2013.6750225

[31] M. Noorafiza, H. Maeda, T. Kinoshita, R. Uda, and M. Shiratori, “Vulnerability analysis using network timestamps in full virtualization virtual machine,” in *Proc. of the 1st Intnl. Conf. on ICISSP*, Angers, France, 2015. [Online.] Available: https://ieeexplore.ieee.org/document/7509933

[32] M. Noorafiza, H. Maeda, and T. Kinoshita, “Virtual machines detection methods using IP timestamps pattern characteristic,” *Intnl. Jour. Comput. Sci. Inf. Technol.*, vol. 8, no. 1, pp. 1–15, Feb. 2016. [Online]. Available: http://aircconline.com/ijcsit/V8N1/8116ijcsit01.pdf

[33] *A Specification of the Internet Protocol (IP) Timestamp Option*,” RFC 781, 1981. [Online]. Available: https://tools.ietf.org/html/rfc781

[34] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn, "Towards sound detection of virtual machines," in *Botnet Detection*, Boston, MA, USA,2008. [Online.] Available: https://link.springer.com/content/pdf/10.1007%2F978-0-387-68768-1_5.pdf

[35] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn, "Remote detection of virtual machine monitors with fuzzy benchmarking," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 3, pp. 83-92, Apr. 2008. [Online]. doi:10.1145/1368506.1368518

[36] M. Athreya, "Subverting Linux on-the-fly using hardware virtualization technology," M.S. thesis, Dept. of Elec. and Comp. Eng., GA Tech, Atlanta, GA, 2010. [Online.] Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.728.453&rep=rep1&type =pdf

[37] C. Thompson, M. Huntley, and C. Link, "Virtualization detection: new strategies and their effectiveness," Univ. of Minn., Minneapolis, MN, USA, 2010. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.7877&rep=rep1&type=pdf

[38] N. Jian, W. Huaimin, G. Shize, and L. Bo, "CBD: A counter-based detection method for VMM in hardware virtualization technology," in *2010 1st Intl. Conf. on Perv. Comp., Sign. Proc. and App.*, Harbin, China, 2010. [Online]. doi: 10.1109/PCSPA.2010.92

[39] R. Patil and C. Modi, "An exhaustive survey on security concerns and solutions at different components of virtualization," *ACM CSUR*, vol. 52, no. 1, pp. 1–38, Feb. 2019. [Online.] doi: 10.1145/3287306

[40] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: cross-VM row hammer attacks and privilege escalation," in *Proc. of the 25th USENIX Conf. on Sec. Symp.*, Austin, TX, USA, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/xiao

[41] J. Rutkowska and A. Tereshkin, "IsGameOver() anyone?," presented at Black Hat, Las Vegas, NV, USA, 2007. [Online]. Available: https://archive.org/details/2007_BlackHat_Vegas-V21-Rutkowska-Tereshkin-Is_Game_Over

[42] T. K. Lengyel and S. Proskurin, "Stealthy, hypervisor-based malware analysis," presented at Hacktivity, Budapest, Hungary, 2016. [Online]. Available: https://www.youtube.com/watch?v=86EvJK2Ef_U

[43] T. K. Lengyel, “Virtual machine introspection to detect and protect,” presented at Hacktivity, Budapest, Hungary, 2014. [Online]. Available: https://www.youtube.com/watch?v=EZPXy314q3E

[44] Intel Inc., “Intel virtualization technology for directed I/O specification,” Rep. D51397-010, 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf

[45] C. H. Hong, I. Spence, and D. S. Nikolopoulos, “GPU virtualization and scheduling methods: a comprehensive survey,” *ACM Comput. Surv.*, vol. 50, no. 3, pp. 1–37, Jun. 2017. [Online]. doi: 10.1145/3068281

[46] R. Shea and J. Liu, “On GPU pass-through performance for cloud gaming: experiments and analysis,” in *2013 12th Ann. Work. on NetGames*, Denver, CO, USA, 2013. [Online]. doi: 10.1109/NetGames.2013.6820614

[47] T. K. Lengyel, “Drakvulf black-box binary analysis system.” Drakvuf, Accessed March 3, 2019. [Online]. Available: https://drakvuf.com/

[48] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, “Compatibility is not transparency: VMM detection myths and realities” in *Proc. of the 11th Work. On HotOS*, 2007. [Online]. Available: https://www.usenix.org/legacy/events/hotos07/tech/full_papers/garfinkel/garfinkel_html/index.html

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California