2020-09

ARTIFACT MITIGATION IN HIGH-FIDELITY HYPERVISORS

Norine, Christopher R.

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ARTIFACT MITIGATION IN HIGH-FIDELITY HYPERVISORS

by

Christopher R. Norine

September 2020

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** September 2020	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE ARTIFACT MITIGATION IN HIGH-FIDELITY HYPERVISORS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Christopher R. Norine	
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) N/A	**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)

The use of hypervisors for cyber operations has increased significantly over the past decade, resulting in an associated increase in the demand for higher-fidelity hypervisors. These hypervisors would not exhibit the markers, or artifacts, that expose the presence of the virtualized environments present in most currently available virtualization solutions. To address this, we present an in-depth examination of a subset of virtualization artifacts in order to design and implement a software solution that will reduce the detectability via mitigation of these artifacts. Our analysis includes performant measures of a bare metal machine, a virtualized machine without our mitigations, and a virtualized machine with our mitigations. The analysis also includes a measure of our implemented system's simulated sensor output. Results of the implementation are analyzed to determine the potential performance impact, the accuracy of our system's simulated output, and whether our mitigation technique is appropriate for extending high-fidelity hypervisors.

14. SUBJECT TERMS high fidelity hypervisors, virtualization, artifact mitigation	**15. NUMBER OF PAGES** 69
	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.

ARTIFACT MITIGATION IN HIGH-FIDELITY HYPERVISORS

Christopher R. Norine
Lieutenant Commander, United States Navy
BA, University of Rochester, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2020

Approved by: Alan B. Shaffer
Advisor

Gurminder Singh
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The use of hypervisors for cyber operations has increased significantly over the past decade, resulting in an associated increase in the demand for higher-fidelity hypervisors. These hypervisors would not exhibit the markers, or artifacts, that expose the presence of the virtualized environments present in most currently available virtualization solutions. To address this, we present an in-depth examination of a subset of virtualization artifacts in order to design and implement a software solution that will reduce the detectability via mitigation of these artifacts. Our analysis includes performant measures of a bare metal machine, a virtualized machine without our mitigations, and a virtualized machine with our mitigations. The analysis also includes a measure of our implemented system's simulated sensor output. Results of the implementation are analyzed to determine the potential performance impact, the accuracy of our system's simulated output, and whether our mitigation technique is appropriate for extending high-fidelity hypervisors.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ARM	Advanced RISC (Reduced Instruction Set Computing) Machine
CPU	Central Processing Unit
DOD	Department of Defense
HFH	High-Fidelity Hypervisor
HVM	Hardware Virtual Machine
IDT	Interrupt Descriptor Table
IDTR	Interrupt Descriptor Table Register
JSON	JavaScript Object Notation
KDBG	Kernel Debugging
KVM	Keyboard, Video, and Mouse
NOP	No Operation
OCO	Offensive Cyber Operations
OS	Operating System
PCG	Platform Compatibility Guide
PID	Process Identification
PV	Paravirtualization
QEMU	Quick EMUlator
RAM	Random Access Memory
SIDT	Store Interrupt Descriptor Table
SOSP	Symposium on Operating Systems Principles
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMX	Virtual Machine eXtensions

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my wife, Samantha, for her patience, support, motivation to persevere regardless of the circumstances, and willingness to juggle two young girls on her own so I could read, write, and code whenever I needed. Without her willingness to help, especially in the middle of a pandemic, I doubt I would be where I am today or accomplish that which I have.

I would also like to thank my advisors, Alan Shaffer and Gurminder Singh, who provided the seed for my thesis and offered their support, guidance, and the occasional reminder that there is always still work to be done.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. RESEARCH PURPOSE

Virtualization has become a common asset in the cyber operations community. From malware analysis and honeypot operations to training environments for testing cutting-edge cyber tools and techniques, virtual machines (VMs) created and managed by hypervisors offer a safe and isolated environment within which to research and test new methods. A downside to operating within virtual machines is that they often lead to artifacts (or markers) that, upon discovery, may allow an observer to realize they are not operating on a bare metal machine. Another downside is that virtual machines do not faithfully replicate the full functionality of the physical computer.

Ingraham et al. [1] described five major categories of hypervisor characteristics—artifacts, behavior, performance, security, and functionality—that can lead to limitations and problems in virtualization. From his research, it is clear that these characteristics must be evaluated for mitigation to achieve high-fidelity virtualization. While most of these characteristics are a byproduct of tighter host-guest integration and proper separation between the host machine and the guest virtual machine, there may be a desire to hide or mitigate virtualization artifacts. The following describes reasons why this is true.

First, malware analysis can greatly benefit from a high-fidelity hypervisor (HFH). For this research, we define an HFH as a hypervisor that is able to present a VM that exhibits behavior in each of the five previously discussed categories to be indistinguishable from a bare metal machine (i.e., a digital twin). Dinaburg et al. [2] described how "malware authors are incentivized to complicate attempts at understanding the internal workings of their creation." These complications include techniques that can be described as anti-debugging, anti-instrumentation, and anti-VM to frustrate would-be analysts and prevent deeper understanding of the malware. Indeed, Chen et al. [3] characterized the prevalence of evasion techniques in modern malware. According to their research, over 40% of the 6,900 total malware samples they examined reduced their malicious behavior whenever a debugger was attached, or when the malware suspected it was executing within a virtual

machine. Artifact mitigation enables an HFH to show no signs of its virtualized environment, allowing analysts to more fully explore the functionality of target malware.

Second, an organization running a honeypot would benefit greatly from an HFH. A virtualized environment is ideal for the execution of a honeypot, therefore malware that encounters such a system will likely attempt to determine if the environment is virtualized or not [3]. An HFH with artifact mitigation would be a better environment for honeypots to operate in, as they would exhibit the behaviors of a bare metal machine without any of the artifacts typically present in virtual machines.

Lastly, it is essential for cyber operators to have a holistic environment in which to develop, test, train, and rehearse their cyber tools and techniques. From an offensive standpoint, it would be impractical to test certain offensive cyber operations (OCO) on a bare metal machine, since the results will likely damage or corrupt these test systems. Recovery will ultimately take time away from the cyber operators, and either reduce the total time spent training and testing or increase the time taken to reach a working solution. Neither scenario is ideal nor desirable. By offering an HFH that is able to present a system's "digital twin," it can be possible to suppress the artifacts that affect feedback to the operator while still providing a target environment that behaves *exactly* as its bare metal equivalent would.

For these reasons, it is essential to examine different aspects that reduce the fidelity of an off-the-shelf hypervisor. In doing so, we need to design and implement mitigation measures that can increase the overall fidelity of hypervisors, while ensuring that the execution of the hypervisors and guest operating systems are not compromised.

B. RESEARCH OBJECTIVE

In this thesis, our research objective is to design and implement a software solution to increase the fidelity of hypervisors by decreasing or eliminating the likelihood of detecting certain virtualization artifacts. In particular, this research is focused on supporting user programs that rely on data generated by sensors embedded in computers, such as a temperature or ambient light sensor. This solution, when encountered by the user, would

produce simulated sensor data that would be consistent with a bare metal system's current operating conditions and environment.

C. RESEARCH QUESTIONS

In our research, we investigate whether it is possible to mitigate virtualization artifacts in a manner that is transparent to a guest VM, while still maintaining the appearance of a bare-metal machine in terms of performance and artifact detectability. The following questions are addressed by this research:

1. Primary Question

What techniques can be implemented within a hypervisor to decrease detectable artifacts present in guest host virtual machines?

2. Secondary Question

How can we apply specific techniques to an open-source hypervisor to increase the overall fidelity of virtual machines managed by that hypervisor?

3. Tertiary Question

How accurately will these techniques replicate a bare metal machine's state and environment during its mitigation of artifacts, and how can we measure this accuracy?

D. BENEFITS OF STUDY

This research will benefit the Department of Defense (DOD)by offering a solution that will enhance the readiness and training of both offensive and defensive cyber operators as well as providing more appropriate testing and production environments for operations executed by the Cyber Warfare community.

E. THESIS ORGANIZATION

This thesis is organized into four additional chapters: background, system design and implementation, system testing, and conclusions and future work.

The next chapter provides a baseline understanding of the various virtualization options available to both the DOD and private industry. It defines key terminology related to virtualization and establishes an unambiguous set of terms and concepts as a foundation for this thesis. It also includes a high-level overview of the different hypervisor solutions available as well as the capabilities and drawbacks of each. Then, it introduces the various types of virtualization artifacts alongside an overview of the methods to detect these artifacts and potentially mitigate them. Lastly, additional software that can be useful, but not directly related to, our research is discussed.

Chapter III outlines the design and implementation of a system dedicated to mitigating VM artifacts with no modification to the guest VM. First, a specific type of artifact is targeted for our system, and a testbed system is described that facilitated our research. Next, a specific subset of VM artifacts is targeted and our mitigation technique is discussed. Lastly, the technical design of our system and the artifact mitigation process is presented.

Chapter IV discusses the methodology of the testing of our implemented system. We present our results with regards to both performance and accuracy by comparing the results of a bare-metal machine to a VM with no mitigation in place, as well as to a VM with the mitigation measures in place. We also examine limitations to our experimentation as well as possible effects of those limitations.

Finally, we present our conclusions in the final chapter, along with suggestions for future work to extend this research.

II. BACKGROUND

A. INTRODUCTION

Since any work in artifact mitigation depends on the underlying hypervisor implementation, it is important to explore the key concepts of hypervisors and the technology enabling them. The multitude of hypervisors and their unique implementations lends a complexity to the subject that requires a review of concepts central to hypervisors and virtualization. By working through the many aspects of virtualization, an appropriate base is set upon which work involving the mitigation of artifacts can be appropriately described and implemented.

B. HYPERVISOR OVERVIEW

The following section is a brief overview of key terminology and principles of hypervisors and their functionality.

1. Terminology

The following section introduces and explains terminology common to hypervisors and discussions of their functionality.

a. Hypervisor

Hypervisors are a specific, special form of system software designed to run virtual machines with low overhead. Typical hypervisors can operate on a single machine or can utilize cloud/distributed resources to support a large number of virtual machines that can be operated concurrently. For the purposes of this thesis, the hypervisor operates on a single machine along with any virtual machines that it is managing. Hypervisors can run at the layer between the hardware and operating system (Type 1), or as user-level applications operating in the user space of an operating system (Type 2). These distinctions are discussed later in this chapter. Hypervisors are also known as virtual machine monitors (VMMs).

b. Virtualization

Virtualization is defined as "nothing more than an instance of layering for which the exposed abstraction is equivalent to the underlying physical resource"[4]. Furthermore, Singh defines it as such:

> Virtualization is 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. [5]

For this thesis, virtualization is defined as the layering of execution environments such that no translation or cross-architectural execution between the guest system and the host system is required to ensure execution on the host system.

c. Emulation

Emulation is "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]. Emulation incurs a significant overhead cost, as the underlying execution environment must translate instructions from the emulated CPU architecture to instructions native to the host CPU architecture. This overhead is not required in a virtualized environment. A common example of an emulator is QEMU (Quick EMUlator), which will be discussed below under Type 2 hypervisors [6].

d. Simulation

Simulation is typically performed in a user-level application that aims to provide a very accurate replica of a given architecture. While the level of accurate execution typically found in simulators would normally make them very desirable, they often come with a slowdown factor of anywhere between 5x and 1000x, depending on the level of detail in the simulation. This constraint makes them undesirable other than in scenarios that prioritize accuracy over speed and usability [4].

2. Technology

The following section briefly describes the underlying technology and functionality found in most hypervisors.

a. Type 1 Hypervisor

Type 1 hypervisors operate directly between the virtualized machine and the hardware. Most Type 1 hypervisors operate as both the host operating system and the virtual machine monitor. This allows them to have full control of the host machine and its resources, and thus they do not need to do additional coordination of system resources with a host operating system. Since Type 1 hypervisors operate at the OS level, there is less overhead compared to operating as an application within an OS, as is the case with Type 2 hypervisors. A few examples of Type 1 hypervisors include The Xen Project, Microsoft Hyper-V, and VMware vSphere.

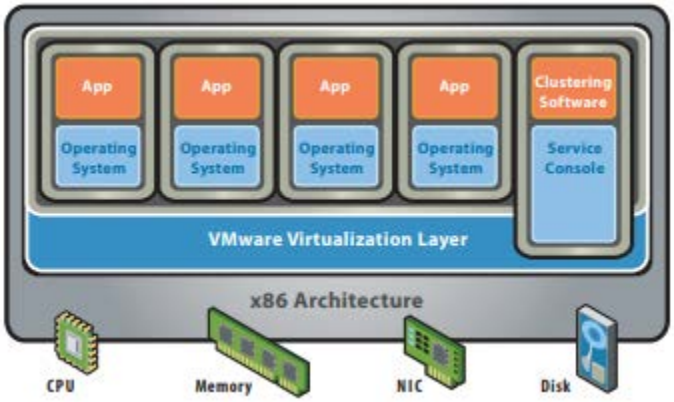

Figure 1. Type 1 Hypervisor. Source: [7].

b. Type 2 Hypervisor

Type 2 hypervisors operate at the application layer, although they have full control of the host machine CPU during execution of the guest OS. Additional overhead is incurred as the host OS and hypervisor execute switches similar in nature to CPU context switches to achieve virtualization [4]. Some popular examples of Type 2 hypervisors include VMware Workstation and Fusion, QEMU with KVM, and Oracle VirtualBox.

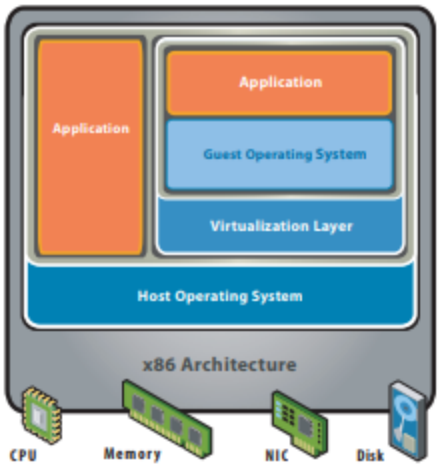

Figure 2. Type 2 Hypervisor. Source: [7].

3. Methods of Virtualization

a. Binary Translation

Binary translation is a form of recompilation which "enables code written for a *source architecture* (or instruction set) to run on another *destination architecture*, without access to the original source code" [8]. The two types of binary translation are static (the program is translated prior to runtime) and dynamic (the program instructions are translated as they are read). Binary translation is generally considered difficult from an engineering point of view as a translator is very specialized and it may not be possible to re-target a given translator to a different architecture without a significant amount of extra work [8].

b. Full Virtualization

Full virtualization (also referred to as hardware virtualization) is where the guest system is unaware of the hypervisor. Instructions that are sensitive or privileged must be caught by the hypervisor without causing issues or being observable inside the virtualized environment. Full virtualization does not require specialized instructions or device drivers but can inflict performance penalties since the hypervisor has to handle sensitive or privileged instructions without impacting the guest system.

c. Paravirtualization

In paravirtualization (PV), the virtualized system is "aware" that it is running within a hypervisor. PV requires specialized kernels and other device drivers that take advantage of communication channels present between the virtual machine and the hypervisor. This significantly reduces the level of overhead required in full virtualization; however this is at the cost of requiring special PV-aware device drivers.

C. POPULAR HYPERVISORS

This next section describes several popular open- and closed-source hypervisors along with a high-level overview of their implementation approaches.

1. Type 1 Hypervisors

The following are various Type 1 hypervisors that are commonly found in-use in both commercial and personal usage.

a. Xen (The Xen Project)

The Xen hypervisor was introduced in 2003 in the Symposium on Operating System Principles (SOSP) paper "Xen and the Art of Virtualization" and is consistently regarded as the best example of an open-source Type 1 hypervisor. As one of the first hypervisors to introduce the concept of paravirtualization, Xen presents a virtualization solution that incurs low overhead compared to other software solutions or virtualization methods. The design team of Xen focused on four major design principles [9]:

1. Running binaries without modification was essential.
2. Supporting full-fledged modern operating systems to allow complex server configurations.
3. Utilizing paravirtualization to the maximum extent to obtain the best performance.
4. Masking the effects of virtualization risked correctness and performance of the virtual machines.

b. Microsoft Hyper-V

Microsoft's Hyper-V is a "hypervisor-based virtualization technology for certain x64 versions of Windows" [10]. Like Xen, Hyper-V is a Type 1 hypervisor where the VMM OS (currently Windows 10) coordinates and manages guest "partitions" that are analogous to virtual machines in other virtualization technologies. Normally, full virtualization is executed unless Microsoft's proprietary "Hyper-V Integration Services" are installed within the guest OS, which bypasses the device emulation layer, allowing guests to execute as paravirtualized guests [10].

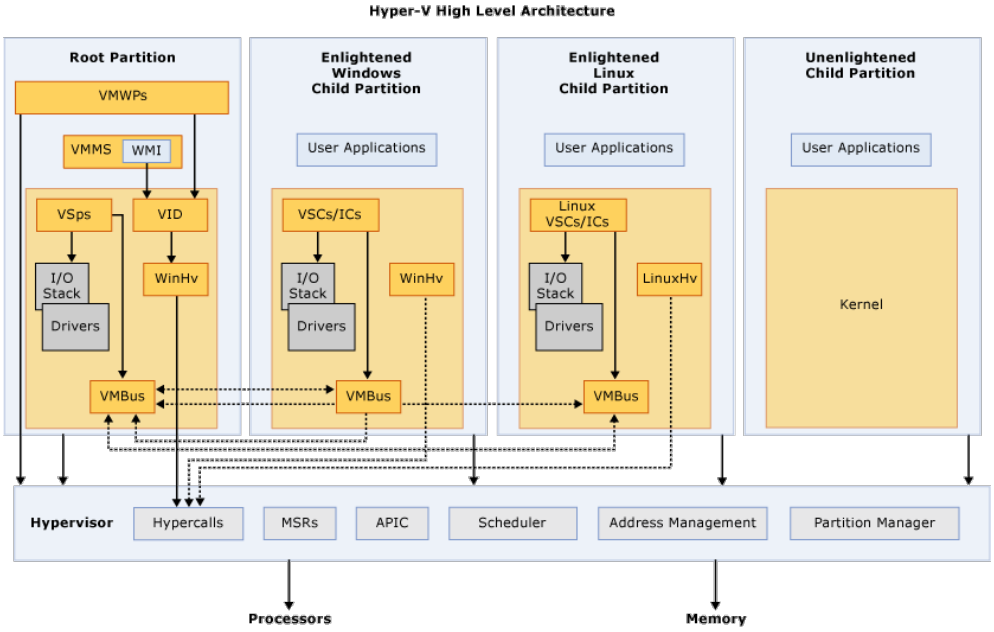

Figure 3. Microsoft Hyper-V Architecture. Source: [10].

c. VMware vSphere / ESXi

ESXi is VMware's proprietary bare-metal hypervisor, intended to be used in conjunction with its vSphere and vCloud products. ESXi offers VMM capabilities across distributed computing resources while offering a web-based user interface, along with various other methods of control. Although available for free with certain editions of VMware vSphere, it is a closed-source software solution.

2. Type 2 Hypervisors

The following are Type 2 hypervisors typically found in commercial and personal settings.

a. VMware Workstation

As VMware's proprietary single-machine Type 2 hypervisor solution, VMware Workstation offers multiple implementations of virtualization, including full virtualization with binary translation, hardware-assisted virtualization, and paravirtualization, utilizing a hosted (by the host operating system) hypervisor. The Workstation hypervisor runs on Windows host systems, while the similar VMware Fusion is designed for MacOS systems. The original VMware Workstation was one of the first platforms enabling x86 virtualization in 1999 [7].

b. Oracle VirtualBox

Oracle's hypervisor solution is an open-source Type 2 hypervisor called VirtualBox. It relies on the hardware virtualization capabilities of the host processor providing either fully virtualized or paravirtualized guest systems. Guest function calls that cause a "VM exit" are captured by the host, processed appropriately, and then control is returned to the guest via "VM entry." VirtualBox also provides multiple paravirtualization interfaces, depending on the guest OS, to increase overall performance while hosting virtual machines [11].

c. QEMU with KVM

The "kernel-based virtual machine" (or KVM) is a Linux kernel module that acts by extending a standard Linux kernel with virtualization capabilities. It accomplishes the task of virtualization via exposed functionality through a character device (/dev/kvm) and by implementing a new operating mode called "guest mode" [12]. These virtual machines can execute natively through a series of system calls to the KVM kernel module and run as individual QEMU processes on the host machine. Without this tie-in to the Linux kernel, QEMU acts as an emulator, as the program on its own does not have virtualization capabilities.

D. HYPERVISOR ARTIFACTS

Virtualization artifacts are markers or indicators of the presence of a VMM or a guest VM. Most of these artifacts fall within one of three categories: service, process, or file system artifacts; random access memory artifacts; and virtualization-specific artifacts, which are further broken down into hardware and capability artifacts. This section will discuss the characteristics and differences between these categories of hypervisor artifacts.

1. Service/Process/File System Artifacts

Most modern hypervisors benefit from the guest OS being aware that it is virtualized. By utilizing paravirtualization and allowing usage of guest-hypervisor communication channels, the overhead usually incurred during full virtualization is reduced or even eliminated. But most hypervisors are also transparent about their PV-specific drivers, which is a large source of artifacts within the guest OS. For example, a VMware Workstation guest running Windows XP with vmtools present (VMware's proprietary PV setup) has over 50 references to "VMware" in the file system and over 300 references in the registry [13]. Although plentiful, these references are not reliable, as researchers have been able to utilize techniques similar to those found in malicious rootkits to readily fool mechanisms looking for these types of artifacts [13].

2. Random Access Memory Artifacts

The following types of artifacts are commonly grouped together as they all involve artifacts that can be discovered through inspection of a VM's random access memory (RAM).

a. Memory References

Hypervisors also insert references to themselves within the guest OS's memory, providing another artifact for possible detection if someone were to dump and search the guest's memory. Researchers discovered over 1500 references to "VMware" within the memory of the guest described in the previous section [13]. It is, however, not trivial or feasible to discover quickly but can be made more effective if the detection mechanism knows which specific segments of memory need to be inspected [13].

b. Pointer Examination

Most modern operating systems utilize tables in memory that are critical to their operation. One example is the Interrupt Descriptor Table (IDT), which holds pointers to various operating system interrupts located within memory [13]. Since the hypervisor and guest both must maintain their own tables, their location within memory cannot be the same, so tools exist (e.g., The Red Pill [14]) that examine the pointer, and determine whether it is operating within a virtual machine or not.

3. Virtualization-Specific Hardware Artifacts

The Linux OS virtual /proc directory can have a wealth of virtualization artifacts. Paravirtualized guests, by definition, utilize virtual device drivers designed to facilitate communication with the hypervisor, along with reducing latency. Within a Linux OS guest, there are multiple references to virtual device drivers, typically found in locations such as: the system's logs, *dmesg* command output, and as files within virtual file directories (like /proc). Windows OS guests are not immune to this phenomenon as the registry also contains multiple device registry keys that reference the hypervisor [13].

4. Virtualization-Specific Capability Artifacts

Paravirtualized guests also contain additional machine language instructions that extend the instruction set of the virtualized hardware. Like the PV device drivers, these instructions are meant to facilitate communication and performance with the hypervisor. VMware and Xen are both examples of hypervisors that extend the instruction set architecture, and tools like VMDetect are designed to attempt to run these expanded instructions. A tool can recognize it is operating within a virtualized environment by the fact that the system does not treat these expanded instructions as errors but will accept and continue operating gracefully [13].

E. HYPERVISOR DETECTION

It is important to understand the techniques and software used to detect the existence of hypervisors if we hope to realize the goal of a higher-fidelity hypervisor. From a security standpoint, it is essential that malware not be made aware of the presence of a

hypervisor, since this could allow unwanted system analysis and also present a whole new attack surface through the VMM [15]. Analysis of these techniques and software can potentially open new avenues of artifact mitigation, thus coming close to realizing a hypervisor that is indistinguishable from a bare metal machine.

1. Detection Techniques

This section describes various high-level techniques that can be utilized to discover the presence of a hypervisor or a VM.

a. Count-Based Detection

At the University of Minnesota, research was conducted to quantify timing artifacts present within various VMMs, as compared to bare metal hardware. Thompson et al. [16] experimented by comparing the ratios of NOP instructions to CPUID instructions executed on various VMMs, discovering detectable differences in behavior that are indicative of a VMM. The underlying implementation utilized the fact that the CPUID instruction is privileged and thus adds additional latency since it must be trapped by the hypervisor and handled before returning control back to the guest [17]. Thompson et al. [16] discovered that even in cases of full virtualization like VMware Workstation, the ratio of instructions executed differed noticeably from the bare metal control and with a baseline understanding of how the system should be performing, detection of a VM is likely.

b. Register Inspection-Based Detection

Research by Robin and Irvine [18] found that processors must meet certain requirements to be considered able to support hypervisors. One of these requirements is that there must be a mechanism in place to automatically signal the hypervisor whenever a guest attempts to execute sensitive instructions. Similarly, a more specific instance of "sensitive instructions" includes those "that read or change sensitive registers and/or memory locations such as … interrupt registers" [18]. They further discovered multiple instructions within the Pentium instruction set that violated this rule, allowing a guest OS access to registers such as the Interrupt Descriptor Table Register (IDTR) which, as

discussed below, may allow an outside observer to recognize that they are operating within a virtualized environment.

2. Detection Software

The following are software implementations of various VM detection techniques that are commonly used to determine if a system is virtualized or not.

a. Red Pill

The Red Pill is a small 4-line program written by Rutkowska [14] that executes the SIDT (Store Interrupt Descriptor Table Register) machine instruction. Since a hypervisor and guest OS must both have an IDT, and the CPU only has a single IDTR, the hypervisor must store the guest's IDTR value somewhere else in memory. The instruction itself is not privileged, so the guest is able to retrieve the relocated address which, regardless of the hypervisor present, is in a different location in memory than a bare metal machine would have it located [14].

b. ScoopyNG

ScoopyNG is a collection of tests written by Klein [19] that probe the same sort of artifacts that the Red Pill examines, while also attempting to run VMware-specific machine instructions to access the hypervisor-guest communication channel. Typically, successful detection of a VM by any of these tests is considered proof enough that the system is running in a virtualized environment [13], [19].

c. VMDetect

VMDetect is another collection of tests meant to expose a hypervisor through use of hypervisor-specific machine instructions [13]. It works by registering its own unique handler for invalid OpCodes, then executes hypervisor-specific (i.e., non-standard) machine instructions [13]. If the unique handler is executed after an invalid machine instruction, then the machine in question is either virtualized using full virtualization and is unaware it is virtualized or is a bare metal machine. This technique is effective for both VMware.

d. Paranoid Fish

Paranoid Fish (also known as Pafish) is a "demonstration tool that employs several techniques to detect sandboxes and analysis environments in the same way that malware does" [20]. Since it is primarily designed to ensure that analysis environments are properly implemented to defeat a piece of malware's detection techniques, it is also effective at evaluating a virtual machine and detecting hardware and software-based artifacts that are present.

F. CURRENT ARTIFACT MITIGATION SOFTWARE AND TECHNIQUES

The following section contains a high-level overview of artifact mitigation techniques and software implementations of those techniques.

1. VMmutate

VMmutate is a proof-of-concept application that attempts to mitigate two common techniques for detecting a VMware hypervisor. First, it modifies the VMX configuration parameters in such a way that it can defeat The Red Pill and portions of the ScoopyNG test [15]. Second, it attempts to alter and/or disable the VMware "magic value," which is a specific value loaded into a CPU register when attempting to call hypervisor-specific machine codes that would normally be invalid. Both modifications combined have the consequence of requiring modification to the hypervisor as well as the paravirtualization tools and drivers.

The drawback to this software tool is that it requires an extensive search and replace operation within the VM disk image, which has the potential to be very large. As well, it is possible to encounter the "magic value" in a non-VMware context, requiring the software to be designed well enough to know which values to alter and which to ignore [13].

2. Hypervisor Configuration Modification

A mitigation technique is the modification of configuration files within the hypervisor to remove artifacts either through obscuration or the breakage of the hypervisor-guest communication channel. For example, Liston and Skoudis [13] discovered several

undocumented configuration options that, when set a certain way, broke the hypervisor-guest communication channel, rendering the hypervisor undetectable through The Red Pill or ScoopyNG. The drawback to this technique is that these modifications are neither documented nor officially supported, thus there is no guarantee that these mitigation techniques will remain effective given that future updates can often break undocumented features [15].

G. OTHER RELATED WORK

Although not designed for detection mitigation, there are software libraries and software tools that were originally intended to examine virtual machines and aid in their analysis, but that could also be used as a means of obfuscating nontrivial artifacts within the guest system or the hypervisor. An example of this might be artifacts present in the Linux */proc* virtual filesystem. Since any mitigation technique implemented within the guest system could be classified as an artifact, by virtue of its presence within the guest filesystem, it is also worthwhile to examine solutions that are employed from outside the virtual machine.

1. LibVMI

LibVMI is an offshoot of the XenAccess Project, which is meant to be a means of virtual machine introspection focused on Xen hypervisors. Specifically, LibVMI aims to be less platform dependent and able to support multiple different hypervisor solutions. It provides a means of monitoring (by reading memory values) and control (by writing new values to memory) from outside the guest virtual machine and is thus able to remain undetected from the perspective of the guest system [21].

2. DRAKVUF

DRAKVUF is an “agentless black-box binary analysis system” designed to utilize LibVMI and the Xen hypervisor to monitor and trace binary execution of a virtual machine from outside the guest itself [22]. It is traditionally used for stealthy malware analysis, but also has the ability to trap specific system calls, giving it the potential for arbitrary data/process injection in Windows guest systems. DRAKVUF also utilizes a plugin-based

system that is much less complicated to utilize, as compared to modifying a hypervisor's source code directly. DRAKVUF currently requires the use of Intel x86 processors to leverage virtualization technology present, but there has also been initial development of an ARM-based version [23].

H. SUMMARY

As the survey of hypervisor technology shows, many different hypervisors are available for implementation, some potentially along the path toward a high-fidelity hypervisor. These hypervisors are designed to operate either directly above the hardware level or as applications within another operating system. The guest systems may also be operated at different "levels" of virtualization, ranging from fully translated hosts that are completely unaware of the hypervisor to paravirtualized hosts that are able to capitalize on communication channels and achieve near-native speed and latency. However, all these implementations create virtualization artifacts, which must be mitigated to prevent identification of virtualization. Although detection techniques have evolved and become better over the last fifteen to twenty years, our goal is to implement techniques that are able to avoid detection by commonly employed hypervisor detection programs.

III. SYSTEM DESIGN AND IMPLEMENTATION

In this chapter, we discuss the detailed design of the system for this research, focusing on the rationale behind specific design choices as well as assumptions made.

A. OVERVIEW

In our research, we have decided to extend the Xen hypervisor that leverages DRAKVUF and LibVMI's introspection abilities to create a plugin named **smokescreen**. Smokescreen works by mitigating device and capability-specific virtualization artifacts present in a VM. It does so by performing introspection on each system call that attempts to execute a program and, if **smokescreen** matches a pre-determined list of artifact-exposing binaries, it replaces the path and redirects the system to execute a modified version of the program to obscure the VM artifact. This obscuration is achieved by having the modified program present seemingly legitimate output in the same fashion as the unmodified program. Since all calls to execute programs are monitored, any calls that do not require redirection are immediately released back to VM execution to minimize performance penalties within the system, which may themselves indicate the existence of a virtual environment.

B. HOST SYSTEM

One of the more important decisions made during our research was the configuration of the test system. Both the hypervisor and the guest system were configured to be representative of a majority of different configurations found on web servers utilized within the last decade.

According to W3Techs, the majority of web servers have operated utilizing some form of Unix-based operating system, with a further breakdown showing that the Ubuntu OS (a Debian-based derivative) was the most likely Linux operating system used [24]. We elected to utilize Ubuntu 16.04.6 LTS since this version continues to enjoy long term support via its owner, Canonical, so it is likely to be encountered "in the wild" quite often. The fact that the DRAKVUF/Xen combination has been tested and confirmed to work on

Ubuntu-based installations reinforced our decision to utilize Ubuntu 16.04.6 LTS as both the Hypervisor (to ensure compatibility with DRAKVUF and Xen) and the guest OS (to ensure a representative system). As far as the guest system was concerned, the only modification required to ensure correct **smokescreen** execution was that any modified binary must reside within the guest OS due to current limitations of the DRAKVUF system.

C. HYPERVISOR

We utilize a Type-1 hypervisor for our system since they are more typical in commercial settings compared to the application-based Type-2 hypervisor. The Xen hypervisor was chosen due to its open-source nature and the fact that it is widely adopted as a hypervisor solution in the tech sector. Additionally, since DRAKVUF and LibVMI were originally designed to work in conjunction with the Xen hypervisor, we were able to leverage that inherent compatibility to achieve a stable platform for our research.

D. DRAKVUF

According to the author and maintainer of the DRAKVUF software suite, it is a "virtualization based agentless black-box binary analysis system" [23]. A key feature of DRAKVUF is its ability to examine and manipulate the execution of arbitrary binaries within a guest operating system without having to install any additional analysis software within the VM. Our end goal, the mitigation of device and capability-based artifacts, was achieved through three separate components found within the DRAKVUF system.

1. Rekall

Rekall is an open-source Python-based framework utilized for "the extraction and analysis of digital artifacts (in) computer systems" [25]. It works by analyzing the currently running kernel on the system to determine the kernel configuration as well as the kernel source headers to map out the locations of essential kernel structures and outputs the locations of these structures as a standardized JSON (JavaScript Object Notation) file. This tailored profile contains key information such as the base memory address of the kernel structure, as well as important address offsets of kernel memory objects and, most

importantly, system call locations. Armed with these memory locations, DRAKVUF is able to leverage LibVMI to trap these system calls.

2. LibVMI

LibVMI is a critical component of DRAKVUF that enables the implementation of Virtual Machine Introspection, specifically allowing access to a live (i.e., running) VM through physical or virtual memory addresses and kernel symbols. The Rekall profile allows LibVMI to "bypass the use (of) the in-memory KdDebuggerData (KDBG) structure normally used by memory forensics tools and thus allows introspecting domains where this structure is either corrupted or encoded (like in the case of Windows 8 x64)" [21]. For our research, LibVMI is the primary means of ensuring that the specific system calls we desire to intercept are correctly identified, located, and properly trapped. We then analyze these system calls and, if modification is determined to be necessary, modify the appropriate parameters. This ensures that any artifact-exposing binaries will be properly intercepted and mitigated through redirection to our modified binaries.

3. Plugin System

DRAKVUF's last essential component to our research involves the powerful plugin system implemented as part of the analysis suite. Through a minor modification of DRAKVUF's source files, we were able to build and integrate various plugins that are initialized and executed during DRAKVUF's main program loop. These plugins are written in C/C++ and utilize a callback function to perform introspection and modification of VMs while they are running. At DRAKVUF initialization these plugins are initialized as well, and the various callback functions are registered to be executed whenever the program's main loop determines it to be appropriate. In the case of our research, these callbacks were configured (via a command-line interface with DRAKVUF) to be executed whenever the desired system calls are executed within the guest operating system's kernel.

E. PLUGIN IMPLEMENTATION

The following section describes the components and implementation of **smokescreen**. We present the hardware and software components followed by a detailed description of the plugin implementation and other implemented software components.

1. Hardware Component Emulation

One particularly interesting application of our research into mitigating device and capability artifacts is the mitigation of artifacts that arise from a VM not having direct access to the host hardware devices (as shown with VM_1 in Figure 4). In most cases, these devices are emulated by the hypervisor to provide a seamless experience to the user, but that is not always the case. For example, most VMs do not have the ability to query physical data such as CPU temperature, or other less often encountered sensors (such as light sensors or motion detectors) that even when present on the host system may not be exposed to the guest system.

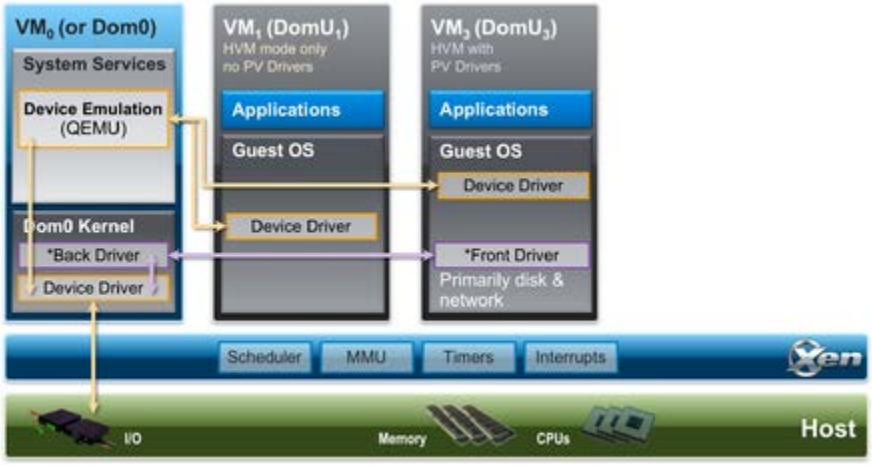

Figure 4. HVM I/O Support. Source [26].

We chose to implement a CPU temperature sensor since it is almost universally found on modern processors but is typically not exposed to guest VMs. To do this, we examined the datasheets of the specific processor used as our testbed (an Intel Core i7-6700) to determine if a temperature model could be ascertained. Upon inspection of the

processor specifications [27], we noted references to the T_{CASE} of the processor. Further inspection of the processor datasheet [28] showed that T_{CASE} corresponds to "Case Temperature [which] is the maximum temperature allowed at the processor Integrated Heat Spreader (IHS)" [27]. It is typically used by designers of cooling systems to ensure that their products are working safely. As shown in Figure 5 of the technical datasheet, the thermal profile is a linear model.

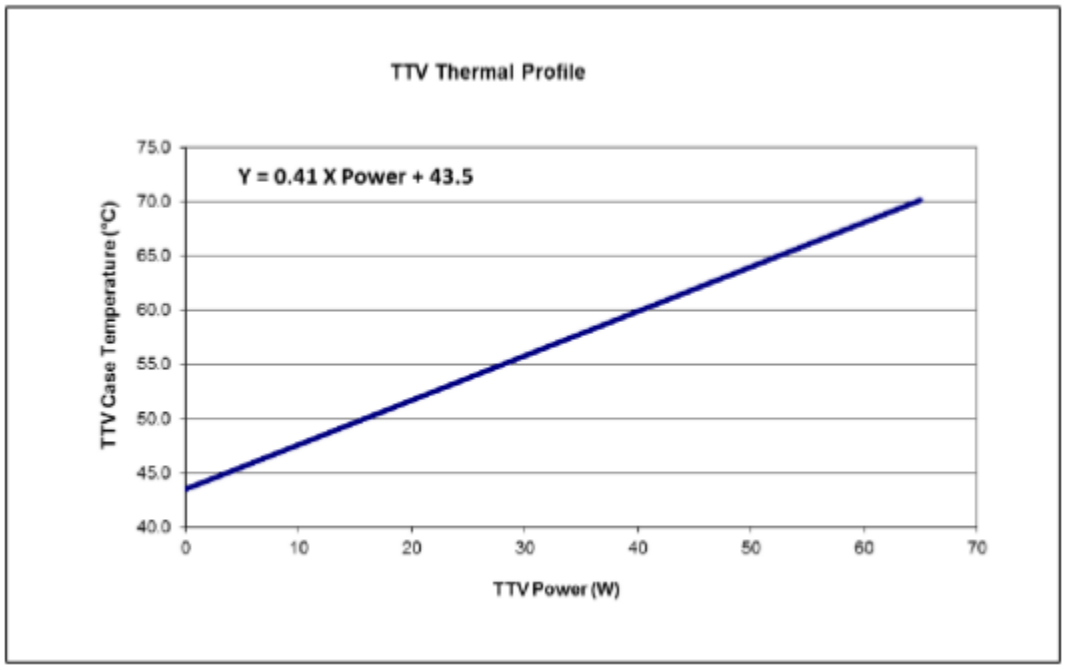

Figure 5. Thermal Test Vehicle Thermal Profile for PCG 2015C Processor (Intel Core i7-6700). Source [27].

From this linear model, we were able to calculate a reasonable CPU temperature profile, based solely on the processor utilization of the guest machine.

2. Software Component

Our system's software component is based on a widely utilized Linux program called *sensors*, from the set of software tools known as *lm-sensors* [29]. As shown in Figure 6, this program works through the examination of system files found in a system's virtual file systems (i.e., */proc* or */sys*). These files contain raw sensor output for various types of

sensors including, but not limited to, temperature sensors. The program collects the data from these sensors, formats it to be human readable, and then outputs it for the user to examine. A VM that does not have access to the underlying system's sensors will report an error condition upon execution of the *sensors* binary since, from the VM's perspective, the sensors do not exist. However, most modern CPUs have a temperature reporting sensor implemented, as it is an essential piece of the architecture's thermal management processes. This lack of reporting on the part of the VM presents itself as both a device and a capability artifact of virtualization.

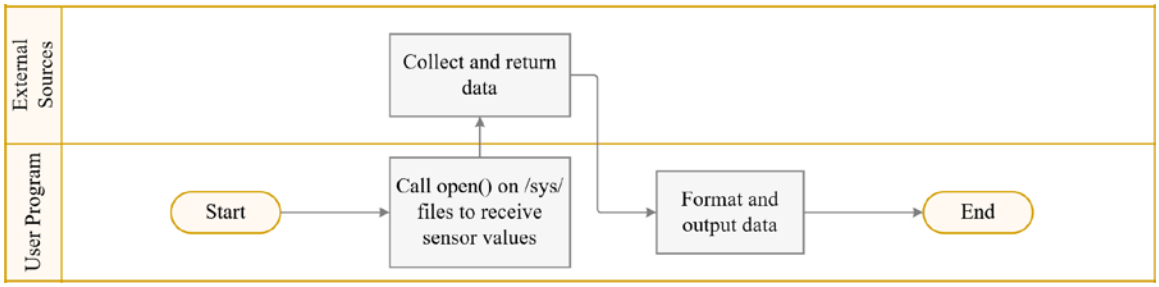

Figure 6. General Execution Flow of *sensors* Binary.

Therefore, we decided to implement a modified version of the *sensors* binary that could report temperature to the end user, as shown in Figure 7. While this could be accomplished simply by having the modified binary output a static value, the goal of implementing a HFH means that the output value should reasonably reflect the current state of the apparent bare metal machine and its environment, even in the context of it being a VM and environment.

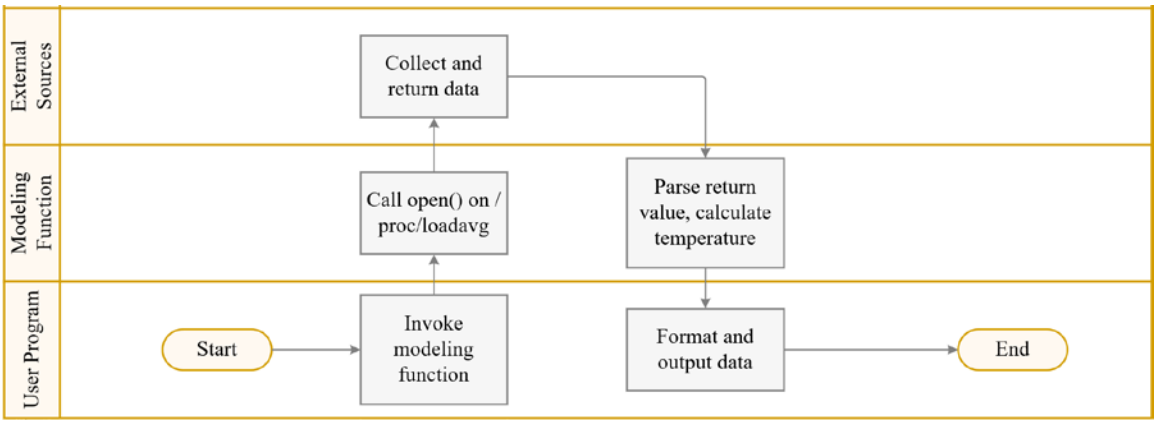

Figure 7. Modified *sensors* Execution Flow.

To this end, our goal was to implement a subroutine or library that, when executed, would sample current conditions within the VM and use those conditions to execute a lookup from the linear temperature model, given system utilization (in watts). To do this, we utilized the T_{CASE} model published by the processor manufacturer, along with the current utilization of the system as determined by the contents of the /proc/loadavg system file. This system file is common to Linux operating systems, and reports running 1-, 5-, and 10-minute processor load averages of the system as floating-point numbers between 0.0 and 1.0 per CPU core/thread (for example, a single core operating at 75% will report a value of 0.75). As shown in Figure 8, by using the 1-minute average as an approximation of current system utilization and the 5- and 10-minute averages to influence the "effect" of cooling solutions (for instance, when longer-term utilization is higher, it is likely that cooling fans have also been working to manage temperature for some amount of time), we were able to implement a simple lookup function that scaled appropriately with system utilization.

```
result = 45.6 + ((POWER*one_min_avg)*0.41);
result -= ((1.0-five_min_avg) + ((1.0-ten_min_avg)*10));
```

Figure 8. C Implementation of the Modeling Function.

3. Means of Implementation

Smokescreen was developed as a plugin for DRAKVUF, primarily implemented in C/C++. DRAKVUF and the parent *plugin* class are implemented in this language so that extension of the class is easier in the same language. For the actions taken by **smokescreen** during execution, knowledge of Sysv 64-bit calling conventions was also required, as memory and register introspection/manipulation require knowledge of CPU registers and their contents prior to kernel system calls.

The plugin itself was implemented as an extension of the *plugin* class (per DRAKVUF requirements) and consists of a total of seven files. Two of the files contain the plugin implementation itself, and one is a header file containing our temperature calculation function. The third file is a patch file for the source files for *sensors.* The fourth and fifth files describe the system calls and executable paths to be trapped and modified, and finally the modified (i.e., recompiled) version of *sensors*.

The plugin files are found in *src/plugins/smokescreen* and consist of *smokescreen.cpp* and *smokescreen.h* per plugin specification. The temperature lookup function consists of a single file *temp_lib.h* and is required to be in the *sensors/lib* folder of the lm-sensors source code to ensure proper recompilation of the *sensors* binary. The patch file must be applied to *sensors/prog/sensors/main.c* of the lm-sensors source code prior to compilation to ensure that our temperature function is called, and to prevent leakage of VM artifacts when the modified *sensors* binary is executed. The system call list is passed as a command line argument to *drakvuf* (via the *-S* flag) which expects an absolute or relative path and can be located wherever the end user desired, and the binary list must be located at */home/xen/xen/binary_list.txt*, but that location is modifiable if required. Finally, the modified *sensors* binary must be located **on the guest VM** at */fake/usr/bin/*, but this location is completely modifiable within the source code for **smokescreen.**

For the implementation of smokescreen, the Linux system call *execve()* requires three parameters: a pointer to the fully qualified path name as a string, a pointer to an argument vector, and a pointer to an environmental variable vector. According to AMD64 Sysv calling conventions, these parameters are found in the CPU registers RDI, RSI, and

RDX, respectively. For our research, the argument and environmental variable vectors are not modified at any time nor require modification for **smokescreen** to execute. Since modification of the argument vector (i.e., the command line arguments) would likely expose our modifications, it is best to leave it untouched as this will ensure that our modified version will preserve functionality not directly related to our emulated sensor such as printing a help or version message. The environmental variable vector is not used directly by the *sensors* program, and modification may lead to system instability.

Our plugin follows a simple execution flow, shown in Figure 9, whenever the callback function is invoked, as described in the following steps:

1. Retrieve and lock (pause) the instance of the VM being monitored.
2. Identify the pathname of the binary about to be executed through introspection.
3. Compare the pathname to a predefined list of fully qualified pathnames to *original* binaries that contain virtualization artifacts.
4. Extract the pathname if a match is found and modify it to be the fully qualified path of our *modified* binary. If not a match, execution will be resumed with no modifications made.
5. Release (un-pause) the VM and allow execution to resume.

We provide further detail on this execution flow in the following.

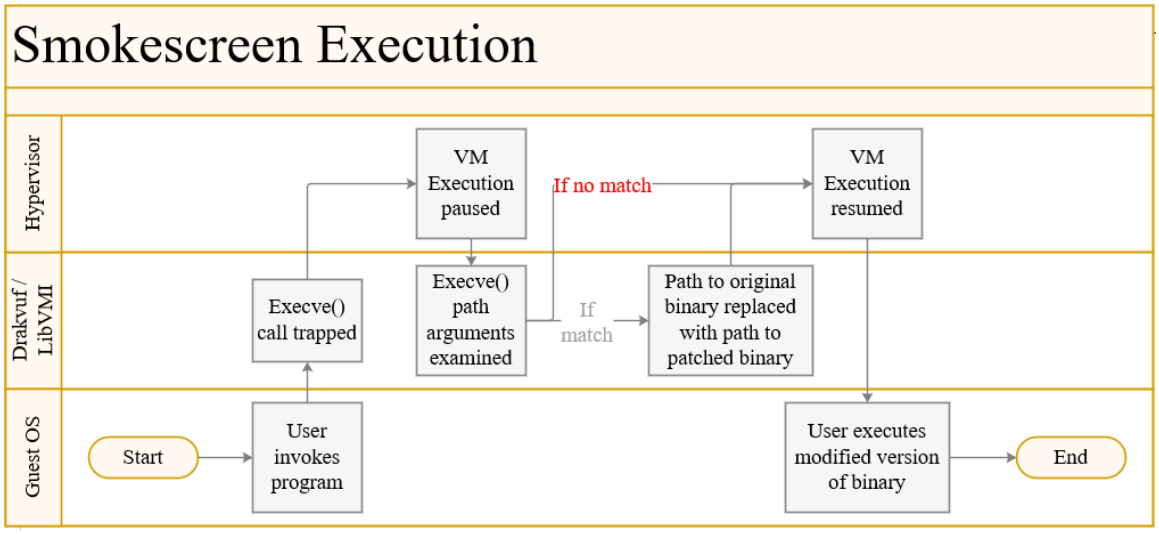

Figure 9. **Smokescreen** Execution Flow

During step 1, the VM instance allows access to, among other things, the values stored within individual CPU registers along with both physical and virtual memory addresses. Since we are concerned with the Linux kernel's *execve()* call, we must be able to extract the pathname from the VM's RDI register for the calling process. As DRAKVUF/LibVMI traps these calls, **smokescreen's** callback function is able to query LibVMI for a *vmi_instance_t* struct named *vmi*. With this struct, in conjunction with the *drakvuf_trap_info_t* struct that is passed to the callback function as a parameter, we are able to extract information from the VM's memory to accomplish our introspection. This extracted information includes data such as the calling process PID (Process Identification) and a complete snapshot of CPU registers and their values for the calling process. When the system call is invoked, RDI contains a pointer to a string representation of the file path to be executed.

As shown in Figure 10, we executed step 2 of our process by utilizing LibVMI's function *vmi_read_str_va()* which returns a pointer to the string containing the fully qualified pathname.

```
printf(“smokescreen activated.\n”);
vmi = drakvuf_lock_and_get_vmi(drakvuf);
pathname = vmi_read_str_va(vmi, info->regs->rdi, info->proc_data.pid);

if (pathname == NULL) {
   drakvuf_release_vmi(drakvuf);
   fprintf(stderr, “[Warning] Captured NULL pathname.\n”);
   return 0;
}
```

Figure 10. Code to Lock the VM and Extract Pathname

Next, as shown in Figure 11, we execute step 3 by comparing our captured pathname and comparing it to our list of pathnames to be intercepted. The list of binaries is loaded at initialization of the plugin and resides outside of the VM.

```
bool found = false;
for (int i = 0; I < s->binaries.size(); i++) {
   printf(“Comparing %s with %s.\n,” s->binaries [i].c_str(), pathname);
   if (s->binaries [i].compare(pathname) == 0) {
       found = true;
       break;
   }
}
if (!found) {
   drakvuf_release_vmi(drakvuf);
   fprintf(stderr, “[Info] No match found.\n”);
   return 0;
}
```

Figure 11. Code to Find a Pathname Match.

If a matching pathname is found, step 4 is executed by constructing a modified pathname and writing it back to the original memory location that still resides in the VM’s RDI register. Otherwise, we immediately release the lock on the VM and return, allowing execution to resume.

From there, the final step is executed by releasing the VM instance, allowing execution to continue in the VM. From the perspective of the executing VM, it is unlikely that this process will be observed unless the user is meticulously monitoring execution of the system in real time.

```
fakepath = (char*) malloc(sizeof(char)*4096);
strcpy(fakepath, “/fake”);
strcat(fakepath, pathname);
printf(“smokescreen attempting to change path to… %s.\n,” fakepath);
vmi_write_va(vmi, info->regs->rdi, info->proc_data.pid,
      strlen(fakepath)+1, fakepath, c);
free(fakepath);
```

Figure 12. Code to Modify Pathname.

F. SUMMARY

In conclusion, we have designed a system that can obfuscate virtualization device and capability artifacts, with minimal impact to the guest VM. By placing modified binaries that mitigate artifacts on our guest VM alongside utilizing VM introspection and DRAKVUF on the hypervisor, we can redirect execution and obfuscate the presence of the modified binaries, as the guest VM believes it is executing the *original* files. Therefore, even if an interested party were to examine the *original* files that are still present, there would be no outward or obvious indications that they were *not* the files being executed. Although the modified binaries are present on the guest VM (thus introducing additional virtualization artifacts), there is no requirement that the modified binaries follow any sort of naming convention which can further obfuscate their presence on the guest VM.

In the next chapter, we present the testing methodology and results of the testing of our system.

IV. SYSTEM TESTING

In this chapter, we present and discuss our methodology for performance and accuracy testing of our system, followed by analysis of their results.

A. TESTING METHODOLOGY

In order to test our system's ability to achieve artifact mitigation, we needed to examine the detectability of VM introspection when redirecting guest VM execution, as well as the accuracy of our sensor data lookup function compared to the bare metal sensor output. We achieved each of these by executing two tests.

1. Performance Testing

First, we tested the performance of our system by examining our modified *sensors* program in three environments: a bare metal machine, a VM where no introspection occurred, and a VM where introspection did occur. Within each environment, we extracted the following data, respectively: the average runtime of *sensors*, the performance overhead of virtualization alone, and the performance overhead of **smokescreen** and its VM introspection. The performance test was considered successful if the introspected VM in our system executed with runtimes similar to those of the other two environments.

To measure the timing of our three environments, we created a Python script to perform multiple system calls that execute *sensors*, both modified and unmodified. For each execution, the individual runtime was calculated and stored. After all test iterations were completed, statistics were extracted from each environment and analysis was conducted to calculate average runtime, standard deviation, and 95% confidence interval for the average. For the purposes of our research, negative timing results were considered invalid and discarded to prevent data skew.

2. Accuracy Testing

We tested the accuracy of our temperature lookup function by building a program that would output our estimate alongside the actual sensor data (i.e., the truth value). More

specifically, we compared the results of our lookup function to the output of the CPU temperature sensor within the */sys* filesystem. Success was indicated if the lookup function produced similar values to the actual temperatures measured while conducting our test.

To measure the values, we created a C program that executed the lookup function then read the actual sensor output. Both values were then redirected to an output collection file. Since our goal is to provide an accurate estimation for the entire range of utilization, we wanted to ensure the widest range of possible values were passed to the lookup function. To accomplish this, NOP loops were executed concurrently with our test program to drive up CPU utilization, simulating the transition from idle to full utilization. The test then halted operations to capture the ramp down to system idle. After testing was complete, the utilization amounts (i.e., output of */proc/loadavg*), estimated temperature (from our function), and actual temperature (from the sensors) were extracted and analyzed to calculate the average deviation between the estimated values and the actual values, both as a raw value (degrees Celsius) and percentage.

B. PERFORMANCE TESTING

In testing the performance of our system, we found it necessary to compare it to both a bare metal machine and a non-introspected VM. With the bare metal machine, we were able to establish a baseline performance, and by including a non-introspected VM, we were then able to determine how much impact from our system could be attributed to virtualization overhead. With these two additional pieces of data, we were able to accurately measure the cost of **smokescreen** in terms of performance.

1. Bare Metal Machine

Our bare metal machine was configured to utilize the same hardware configuration and the same Ubuntu 16.04.6 LTS distribution that were used for the virtual machine. *Lm-sensors* was installed and executed with no modification to the *sensors* program. Several sensors, including the processor temperature sensor, were detected out-of-the-box (i.e., without additional configuration required).

In testing the bare metal machine, as the results show in Table 1, we were able to establish that *sensors* executed for an average of 2.886ms, with most values falling within 0.5ms of that time. Based on the number of iterations executed, we were able to establish a 95% confidence interval of +/-0.00986ms, making us very sure about the accuracy of our average execution time.

Table 1. Bare Metal Timing Results.

Total Iterations	9,944
Average Execution Time	2.886ms
Standard Deviation	0.5ms
95% Confidence Interval	+/- 0.00986ms

2. Hypervisor without Mitigations Present

To establish the cost of hypervisor overhead, we needed to measure the performance difference between an introspected and a non-introspected VM. To do this, we used the Dom0 VM, which is the VM that acts as our user interface to the Xen hypervisor directly (i.e., the environment that DRAKVUF runs in while introspecting a guest VM). We chose the Dom0 VM, as opposed to our guest VM, for two reasons: first, the Dom0 VM is defined by the Xen hypervisor as being another virtual machine that has elevated access to the hardware level, which is not available in the guest VM [9]. Although the Dom0 VM does not have access to the CPU temperature sensor, it does have access to other sensors (which our guest host does not) and it still provides a similar flow of execution for the unmodified *sensors* program when compared to the bare metal machine. Second, our test system was designed intentionally to ensure that the VMM and the guest VM were executing identically-configured kernels, and that both had access to similar resources through the hypervisor, providing a nearly identical environment when compared to the guest VM.

The performance of this test system is shown in Table 2. We see that execution time increased by approximately 1.2ms on average over the bare metal system, and resulted

in a wider standard deviation. Due to the increased standard deviation, our 95% confidence interval with regards to execution time increased to +/-0.01425ms.

Table 2. Virtual Machine Timing (No Introspection) Results.

Total Iterations	9,953
Average Execution Time	4.022ms
Standard Deviation	0.73ms
95% Confidence Interval	+/- 0.01425ms

3. Hypervisor with Mitigations Present

Finally, the guest VM was tested with DRAKVUF running on the hypervisor, with **smokescreen** implemented and executing as a part of the DRAKVUF instance. The guest VM also executed the modified version of *sensors*. As shown in Table 3, execution times for this system increased by a relatively large amount over the other test systems, with average execution time increasing by 5.964ms over the bare metal system, and by 4.828ms over the non-introspection system. In addition, this performance resulted in a wider range of observed execution times, as seen by the higher standard deviation of 1.493ms. Here we experienced the widest of our 95% confidence intervals, with a value of +/-0.04816ms. Although this is significantly larger of an interval compared to the other two environments, it still shows that we can reasonably expect execution to fall within a small range of possible times.

Table 3. Virtual Machine Timing (with Introspection) Results.

Total Iterations	9,959
Average Execution Time	8.850ms
Standard Deviation	1.493ms
95% Confidence Interval	+/- 0.04816ms

C. ACCURACY TESTING

Another important metric in determining success in our system is whether our temperature function accurately reflects the bare metal system's state without direct access to the sensors. To test this, we compared our lookup function output to the bare metal system's temperature sensor raw output, as stored in the */sys* folder (in Ubuntu 16.04.6 LTS). The results are broken down into three categories as shown in Table 4. Overall, we saw consistent underestimation, with our lookup function typically returning values around 13°C, or 16.28%, lower than the raw sensor values. By also considering the calculated standard deviation of the differences, most estimation differences were anywhere between -22.3917°C and -4.0739°C too low. When broken down by utilization transition, the idle to full transition saw much larger underestimations up to 35°C with an average difference of 21.9% between the estimated and actual values. Underestimations seen here typically fell within a range of -20.503°C and -11.799°C. The full to idle transition saw both under and overestimations with deviations between -15°C and +17°C, but an average difference of only 0.52% and most differences falling within between -3.4256°C and +2.7884°C.

Table 4. Lookup Function Accuracy Results.

Overall Results (3600 data points)	
Average Difference	-13.2328°C
Standard Deviation	9.1589°C
Average Difference Percentage	-16.28%
Largest Underestimation	-35.000°C
Largest Overestimation	17.000°C
Idle to Full Utilization Transition (2403 data points)	
Average Difference	-17.8244°C
Standard Deviation	4.3521°C
Average Difference Percentage	-21.94%
Largest Underestimation	-35.000°C
Largest Overestimation	N/A*
Full to Idle Utilization Transition (1196 data points)	
Average Difference	-0.3186°C
Standard Deviation	3.1070°C
Average Difference Percentage	0.51%
Largest Underestimation	-15.000°C
Largest Overestimation	17.000°C

*No overestimations occurred.

D. ANALYSIS OF RESULTS

In this section, we analyze the results of the performance and accuracy testing, and present possible explanations of these results.

1. Performance Analysis

Initial examination of the performance statistics shows that there is a measurable impact of both virtualization and VM introspection. A histogram of the three test systems' execution times is shown in Figure 13. In Table 2 we saw an increase of ~1.1ms average execution time (from 2.886ms to 4.022ms), a roughly 39% increase, in a virtual machine over a bare metal machine. We also saw an increase in the range of values encountered within one standard deviation. It is possible that this increase in execution time was due to resource sharing between the various VMs as well as context switching (for example, due to sensitive or privileged instructions that much be trapped) between the hypervisor and guest VM that is required for normal virtualization operations.

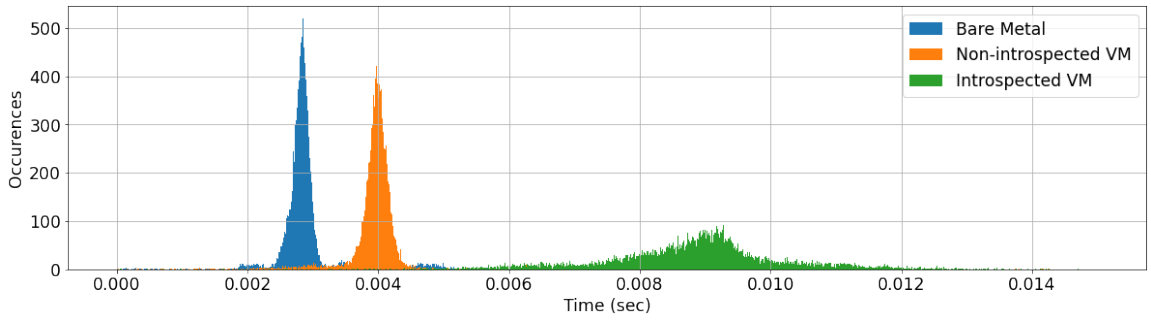

Figure 13. System Performance Results.

Comparing the results of the guest VM alongside **smokescreen** operating, we saw a relatively large increase in the execution time for *sensors*. The average execution time was 8.850ms with a standard deviation of 1.493ms. This difference in timing compared to the bare metal system can likely be attributed to not only the overhead incurred by virtualization, but also to the processing time of **smokescreen** and the lookup function executed by *sensors.* Since the results from the previous two environments show that *sensors* takes approximately 2.886ms to execute, and virtualization overhead added an

additional 1.136ms, then the 8.850ms execution time we observed shows that **smokescreen** increased average execution time by 4.828ms compared to an unmodified VM and increased by 5.964ms compared to a bare metal system. These times correspond to a roughly 83% and 206% increase in average execution time compared to the non-introspected VM and the bare metal machine respectively.

2. Accuracy Analysis

In comparing our lookup function to the actual sensors on the bare metal system, our modelling function followed the overall trend of the actual sensor data. As temperature sensor readings increased, our function returned steadily increasing results, and when the actual temperature decreased, our function also showed a decrease in returned values.

Although our function produced a similar temperature curve to that of the actual sensor data, it consistently underestimated the actual temperature values, as can be seen in Figure 14. This was likely due to a combination of the use of the T_{CASE} temperature as an upper limit to calculatable temperatures, along with the influence of the five- and ten-minute averages on the calculated value. Although T_{CASE} is described by Intel as "the maximum temperature at the integrated heat spreader" [27], it should not be considered an upper limit to temperatures that could be encountered in reality. Next, the intention of the five- and ten-minute averages was to estimate the effect of CPU cooling over time, but as average CPU utilization grew and plateaued, those averages capped the estimated temperature even lower than T_{CASE} (71° C in the case of our Intel i7-6700).

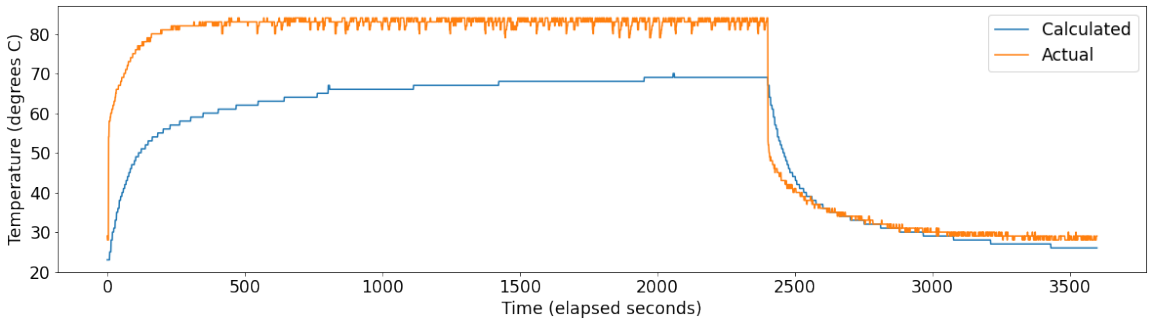

Figure 14. System Accuracy Results.

E. TESTING LIMITATIONS

The version of DRAKVUF that we used limited introspected VMs to a single virtual CPU per VM (which is assigned to a single physical CPU thread by the hypervisor). Additionally, the output of */proc/loadavg* is a combined total of the load average for all present CPU threads, so a 4-core/8-thread CPU (such as the Intel i7-6700) would return values ranging from 0.00 to 8.00 (i.e. 0% to 800%) to signify no utilization on any core to full utilization on all cores, respectively. Since our design assumed access to only a single CPU thread from the start, it did not account for the presence of multiple cores or multiple threads. However, our temperature accuracy testing was performed on a bare metal machine that had access to all eight processor threads in order to have actual sensor data. To account for this, we modified our temperature lookup function for testing as follows:

1. Multiple NOP loops were utilized to ensure 100% utilization across **all** CPU threads.

2. The */proc/loadavg* output was normalized to a range of 0.0 to 1.0, producing values that would be expected by **smokescreen.**

By ensuring that all CPU threads were operating at near-identical utilization, and by normalizing the load averages to that of a single core processor, we were able to produce the range of values that **smokescreen** would expect and would accurately reflect the results produced by the plugin.

F. SUMMARY

In this chapter, we presented the testing methodology and results of our implemented system, **smokescreen**. From a performance point of view, the introduction of our system onto a VM resulted in an average execution time increase of 5.964ms, representing a 206% increase over a baseline bare metal machine. Of that time, 38% of the increase can likely be attributed to the virtualization overhead, and an increase of 167% that can be attributed to **smokescreen**.

From an accuracy point of view, our **smokescreen** system closely followed the temperature curve of the bare metal machine sensor data, however our lookup function

consistently underestimated the CPU temperature. This was likely because of the use of T_{CASE} as a maximum temperature value combined with the usage of higher utilization over time (the five- and ten-minute load averages) acting to further depress on our estimated values.

In the next chapter, we present our conclusions as well as recommendations for future work in VM artifact mitigation.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND FUTURE WORK

At the beginning of our research, we set out to answer three important research questions. We sought to 1) identify techniques utilized to mitigate hypervisor artifacts, 2) design and implement chosen techniques that would mitigate these artifacts, and 3) ensure that our mitigations could accurately estimate the actual state of the underlying system, without access to the sensors that describe that state. Our mitigation tool, **smokescreen**, answers these questions, providing artifact mitigation with a performance cost measured in milliseconds, while being able to provide sensor data that follows the temperature curve of a bare metal system during normal use.

In this chapter, we present our conclusions as well as recommendations for follow-on research and other future work.

A. CONCLUSIONS

In this thesis we designed and implemented **smokescreen** as a DRAKVUF plugin with the goal of mitigating capability and device artifacts common within modern VMs. In particular, we were able to modify a VM's system calls to replace the path of specific programs which produce virtualization artifacts and replace them with modified versions that did not exhibit those artifacts. The artifacts are mitigated by having the modified program estimate the system's state through use of other system information that is present. Although other solutions exist that also provide increased fidelity [30], our goal was to implement a solution that existed mostly outside the guest VM in order to achieve our view of a high-fidelity hypervisor. Our results indicated that **smokescreen** provides increased fidelity but at the expense of increased execution times and potentially introducing other VM artifacts in order to mitigate those we targeted in our system.

1. Artifact Mitigation

By utilizing DRAKVUF [22], we were able to keep most of our software solution outside of the guest VM. The implementation showed that it is possible to redirect execution of some artifact-leaking programs in a way that is difficult to detect, even when

examining execution from within the VM. This results from the fact that the guest VM has no potential indicators that the original program is not being executed.

However, **smokescreen's** implementation and its interactions with LibVMI's API started to make the limitations of VM introspection more apparent. Through **smokescreen** we were able to access CPU register and memory contents, but the plugin could not directly access devices or files in order to modify them. As a result of this, our implementation needed to introduce additional file system artifacts into the guest VM. Since our modifications to the guest VM were limited to data in or pointed at by CPU registers, we were unable to directly manipulate the programs which cause artifact leakage and must instead redirect execution to our own modified versions of the programs *which reside within the guest VM* to mitigate potential leakage. Similarly, since we were unable to manipulate the system files that contain sensor values, we were forced to ensure that our modified programs also calculated the estimated value *inside the guest VM* rather than doing so from without and passing that data to the original program.

2. Performance

Overall impact to performance is an important consideration when determining whether **smokescreen** would be an appropriate building block for an HFH. Our solution's cost to performance is a **206% increase** in the execution time of the *sensors* program. However, when considered in terms of the actual amount of time to run (roughly 3ms to 9ms), the actual execution time would be unlikely to raise suspicions of the presence of a hypervisor, as that difference could easily be attributed to other causes, such as resource sharing among processes, context switches, or other high-priority processes preempting these programs during execution. However, as the number of VMs present on a machine increase (increasing the demand on system resources) or as the number of programs that require redirection increase, it is possible that the introspections could cause a more noticeable system slowdown over time. This could be a limiting factor for deployment of our solution.

3. Accuracy

The accuracy of our system is also an important consideration for deployment within an HFH. Overall, our solution was able to generally follow the temperature curve of the actual CPU temperature, but typically underestimated the values by around **16°C on average**. However, when examined by the utilization trend (idle to full and full to idle), it was seen that almost all of the significant underestimations occurred during the idle to full transition, with the estimations staying relatively close (within 3°C on average) during the full to idle transition. With additional research, a more representative linear model can be created and applied, resulting in a temperature curve that not only follows the correct temperature curve, but is also more accurate in its estimations.

B. LESSONS LEARNED

During our research and implementation of **smokescreen**, we encountered a few teachable moments. First, implementation of a system that operates outside a guest VM but affects operation within that VM can be problematic. DRAKVUF and LibVMI are able to make access to CPU registers and memory contents possible, but only when system calls are executed. From a semantic point of view, we were required to bridge the gap between the high-level *execve()* call and the low-level view presented during introspection. Intimate knowledge of assembly code and the relationship between a process's virtual memory space and the VM's system memory is required in order to correctly (and safely) manipulate that memory during introspection.

Also, it is important for an appropriate amount of research to be conducted to ensure models being implemented, such as CPU temperature, are able to accurately reflect the actual state of the bare metal machine, in both general curve of the values and accuracy of those values.

C. FUTURE WORK

In this section, we present potential future work that may augment or improve **smokescreen's** mitigation techniques as well as for high-fidelity hypervisors overall.

1. Process Injection

At the present time, DRAKVUF is limited in how it can allow manipulation of memory during VM introspection. While manipulation is possible, it can only occur when system calls are executed and trapped through LibVMI. However, a future capability of the system under development called *Process Injection* could enable plugins (like **smokescreen**) to replace the contents of a process's memory space with that of a different process which exists *outside the guest VM*. Once this capability has been implemented, **smokescreen** will act as a natural building block, where our modified programs are able to exist *outside the guest VM* and remove the requirement of having the modified programs present within the guest VM. This would eliminate the file system artifacts, in particular the modified programs, that were introduced by **smokescreen's** current implementation.

2. Mitigation of Other Types of Artifacts

Another limitation of **smokescreen** is that it is only designed to mitigate device and capability artifacts that are present in common programs found within virtual machines. As described earlier, there are other types of artifacts, such as service, process, and file system artifacts, and random access memory artifacts. A system implemented to be an HFH will ultimately need to include mitigations for all these other types of artifacts.

3. Additional Characteristics of High-Fidelity Hypervisors

As described in Chapter I, Ingraham et al. [1] initially described five categories of characteristics of HFHs: artifacts, behavior, performance, security, and functionality. **Smokescreen's** mitigations are focused on increasing the overall fidelity of a hypervisor's characteristics with regards to artifacts. The other four categories will also require additional research in order to develop a hypervisor that is truly an HFH.

LIST OF REFERENCES

[1] C. Ingraham, A. Shaffer, and G. Singh, “High-fidelity virtualization for cyber operations,” in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec. 2019, pp. 196–201, doi: 10.1109/CSCI49370.2019.00040.

[2] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: Malware analysis via hardware virtualization extensions,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2008, pp. 51–62, doi: 10.1145/1455770.1455779.

[3] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, “Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, Jun. 2008, pp. 177–186, doi: 10.1109/DSN.2008.4630086.

[4] E. Bugnion, J. Nieh, and D. Tsafrir, “Hardware and software support for virtualization,” *Synth. Lect. Comput. Archit.*, vol. 12, no. 1, pp. 1–206, Feb. 2017, doi: 10.2200/S00754ED1V01Y201701CAC038.

[5] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, “Supporting soft real-time tasks in the Xen hypervisor,” in *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, New York, NY, USA, 2010, pp. 97–108, doi: 10.1145/1735997.1736012.

[6] F. Bellard, “QEMU, a fast and portable dynamic translator,” *USENIX Annu. Tech. Conf.*, vol. 41, pp. 41–46, Apr. 10, 2005.

[7] “Understanding full virtualization, paravirtualization, and hardware assist.” https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html (accessed Oct. 09, 2019).

[8] S. Bansal and A. Aiken, “Binary translation using peephole superoptimizers,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 177–192, Accessed: Jan. 20, 2020. [Online]. Available: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/bansal/bansal.pdf.

[9] P. Barham *et al.*, “Xen and the art of virtualization,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2003, pp. 164–177, doi: 10.1145/945445.945462.

[10] S. Cooley, H. Juarez, and J. Terry, “Hyper-V architecture.” Microsoft Docs, January 10, 2018. [Online]. Available: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/hyper-v-architecture

[11] Oracle, “Oracle® VM VirtualBox®.” (accessed Jan. 14, 2020). [Online]. Available: https://www.virtualbox.org/manual/

[12] Jun Zhang, Kai Chen, Baojing Zuo, Ruhui Ma, Yaozu Dong, and Haibing Guan, “Performance analysis towards a KVM-based embedded real-time virtualization architecture,” in *5th International Conference on Computer Sciences and Convergence Information Technology*, Nov. 2010, pp. 421–426, doi: 10.1109/ICCIT.2010.5711095.

[13] 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

[14] J. Rutkowska, “Red Pill... or how to detect VMM using (almost) one CPU instruction,” The Invisible Things, Nov. 2004. [Online]. Available: http://web.archive.org/web/20110726182809/http://invisiblethings.org/papers/redpill.html

[15] M. Carpenter, T. Liston, and E. Skoudis, “Hiding virtualization from attackers and malware,” *IEEE Secur. Priv.*, vol. 5, no. 3, pp. 62–65, May 2007, doi: 10.1109/MSP.2007.63.

[16] 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

[17] 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

[18] J. Robin and C. Irvine, “Analysis of the Intel Pentium’s ability to support a secure virtual machine monitor,” Defense Technical Information Center, Fort Belvoir, VA, Aug. 2000. doi: 10.21236/ADA423654.

[19] T. Klein, “trapkit.de - ScoopyNG.” Trapkit, Accessed Jan. 14, 2020. [Online]. Available: http://www.trapkit.de/tools/scoopyng/index.html

[20] A. Ortega, “a0rtega/pafish,” *GitHub*. Accessed on Jan. 14, 2020. [Online]. Available: https://github.com/a0rtega/pafish

[21] GitHub, “libvmi/libvmi: The official home of the LibVMI project is at https://github.com/libvmi/libvmi.” Accessed Feb. 04, 2020. [Online]. Available: https://github.com/libvmi/libvmi

[22] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, “Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, New York, NY, USA, Dec. 2014, pp. 386–395, doi: 10.1145/2664243.2664252.

[23] T. K. Lengyel, “DRAKVUF™ Black-box Binary Analysis System.” DRAKVUF, Accessed on Feb. 04, 2020. [Online]. Available: https://drakvuf.com/

[24] W3Techs, “Usage of web servers broken down by operating systems.” Accessed Jun. 15, 2020. [Online]. Available: https://w3techs.com/technologies/cross/web_server/operating_system

[25] Rekall Forensics, “Rekall Forensics.” Accessed Jun. 19, 2020. [Online]. Available: http://www.rekall-forensic.com/

[26] Xen Project, “Xen project software overview - Xen.” Accessed Jun. 23, 2020. [Online]. Available: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview#HVM_I.2FO_Support

[27] Intel, “Intel® Core™ i7-6700 Processor (8M Cache, up to 4.00 GHz) product specifications.” Accessed May 09, 2020. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/88196/intel-core-i7-6700-processor-8m-cache-up-to-4-00-ghz.html

[28] Intel, “6th Generation Intel® processor families for S-Platforms, datasheet, volume 1 of 2.” Accessed: May 09, 2020. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/332687.

[29] GitHub, “lm-sensors/lm-sensors.” Accessed Jun. 19, 2020. [Online] Available: https://github.com/lm-sensors/lm-sensors

[30] Y. Zhang, F. Xie, Y. Dong, G. Yang, and X. Zhou, “High fidelity virtualization of cyber-physical systems,” *Int. J. Model. Simul. Sci. Comput.*, vol. 04, no. 02, p. 1340005, May 2013, doi: 10.1142/S1793962313400059.

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