NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PROTECTING COMPROMISED SYSTEMS WITH A VIRTUAL-MACHINE PROTECTION AND CHECKING SYSTEM USING OUT-OF-GUEST PERMISSIONS

by

Alexis Peppas

December 2017

Thesis Advisor: Geoffrey G. Xie
Second Reader: Charles D. Prince

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 2017	Master's Thesis MM-DD-YYYY to MM-DD-YYYY

4. TITLE AND SUBTITLE

PROTECTING COMPROMISED SYSTEMS WITH A VIRTUAL-MACHINE PROTECTION AND CHECKING SYSTEM USING OUT-OF-GUEST PERMISSIONS

5. FUNDING NUMBERS

6. AUTHOR(S)

Alexis Peppas

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

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 document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

test

14. SUBJECT TERMS

15. NUMBER OF PAGES 77

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

PROTECTING COMPROMISED SYSTEMS WITH A VIRTUAL-MACHINE PROTECTION AND CHECKING SYSTEM USING OUT-OF-GUEST PERMISSIONS

Alexis Peppas
Lt, Navy
B.S., Hellenic Naval Academy, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2017

Approved by: Geoffrey G. Xie
Thesis Advisor

Charles D. Prince
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

test

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

NIST	National Institute of Standards and Technology
OS	operating system
VM	virtual machine
CPU	central processing unit
VMI	virtual machine introspection
ACL	access control list
SACL	shadow access control list
VMPCS-OGP	Virtual-Machine Protection and Checking System Using Out-Of-Guest Permissions
API	application program interface
VT	virtualization technology
PT	page table
EPT	extended page table
GMFN	guest machine frame number
MFN	machine frame number
IOMMU	input/output memory management unit
GVA	guest virtual address
MAC	mandatory access control
HAP(2)	high assurance process
OI	object identifiers

SGX	software guard extensions
IDS	intrusion detection system
HIDS	host intrusion detection system
NIDS	network intrusion detection system
NIC	network interface card
IoT	internet of things
SCADA	supervisory control and data acquisition
PAM	pluggable authentication module
PWD	present working directory
APT	advanced persistent threat
HAP(1)	hardware assisted paging
UID	userid
GID	groupid

Executive Summary

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

System virtualization, which has been increasing in popularity over the last years, makes it possible to run many and different operating systems (OSes) on the same physical machine. virtual machines (VMs), are run independently of each other on the same physical machine, known as a host, without any indication that there is another OS running on the same host. The software that facilitates this resource sharing capability is called a hypervisor.

The emergence of cloud computing has increased the need for many new and different services from different global vendors. According to the National Institute of Standards and Technology (NIST) [1], "cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Virtualization solved the increasing requirement of resources for these services to run on. Instead of having many separate physical machines running the required different software, usually resulting in underutilization, one machine, with better technical specifications and capabilities like more memory capacity and multi-core central processing units (CPUs), was used, and with virtualization, each vendor could run his services on a dedicated VM.

In order to improve network security on these machines, as well as redundancy among different services, instead of having one VM running all the services required by one vendor, service providers started using many different VMs, each requiring fewer resources and running one or just a couple of services. This way, if one VM fails, the rest of the services keep running. Furthermore, having each VM run only a few services significantly reduces the attack surface available for possible vulnerability exploitation.

This increase in the use of virtualization has driven hardware manufacturers, like Intel and AMD, to introduce special virtualization CPU instructions, to facilitate better, more reliable and secure allocation, sharing, usage and performance of VMs.

1.1 Problem Statement

When an OS runs directly on a physical machine, it allocates and uses its resources to protect itself from network or other types of attacks. But, when it runs on a virtualization platform, the hypervisor stands between the hardware and the running software, and can see what is happening inside a VM.

Despite the evolution of CPU virtualization instructions, and the continuous development of more efficient and secure hypervisors, the bottom-line remains the same. A VM is still a system, with all the vulnerabilities of its running OS and software, and at some point in time, it will be the victim of a successful exploitation.

The native Linux file permission system, although simple to manage and efficient, lacks fine-grained user/group access to files. Once users belong to a group, nothing prohibits them from accessing all the files accessible to that group. Furthermore, when attackers gains access to a system, they will usually try to escalate their privileges by having access to the root account. From that point there is nothing out of reach; the attacker has unrestricted access to the entire system and is free to read and modify files and change the system's configuration to his liking, to serve his purposes.

Garfinkel et al [2] introduces a new technique which leverages the hypervisor's viewing ability. virtual machine introspection (VMI) is the "approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it." Outside in this context means that the inspecting application resides outside the monitored VM and can access the VM's state through the hypervisor. Because a system will be eventually subverted, we want to leverage the introspection capability of a hypervisor to try to protect critical files for the OS, the user, or both. We want to create an out-of-guest access control list (ACL), which we call shadow access control list (SACL), for managing file access inside a VM. We call this mechanism Protecting Compromised Systems with a Virtual-Machine Protection and Checking System Using Out-Of-Guest Permissions (VMPCS-OGP).

In our research, we developed a prototype for a file access monitor and control outside a VM. We used a 64-bit Ubuntu OS running on top of a Xen hypervisor. The prototype leverages the VMI capability of the Xen hypervisor leveraged with the LibVMI application program interface (API) [3], as well as DRAKVUF [4], a system used for dynamic malware

analysis. It includes a modified DRAKVUF implementation, as well as prototypes of the ACL kept on the hypervisor, to be enforced on the guest VM. Our approach is to provide a more strict environment for file access.

In this work we tried to assess how we can leverage the introspection capabilities of the Xen hypervisor to improve the confidentiality, integrity and availability mechanisms built into the OS. Some of these cases include denying access to the root user, who has access to the entire filesystem. We want to make a more fine-grained access control to fill the gap of the Linux native permission bits by denying access to files on users that belong to a group with access. Furthermore, we want to alter the user permissions by keeping a SACL. Moreover, we will try to enforce append-only permission instead of generic write for specific cases of files, which include primarily log files, as we would like to prevent a malicious action to be removed from any logs, as part of covering the tracks of the malicious activity.

This solution can potentially be used in a variety of platforms like internet of things (IoT) or supervisory control and data acquisition (SCADA) systems, cellphones, cloud solutions, essentially everything that runs on a virtualized environment. It could also be used to enhance the filesystem security of end-of-life systems that do not receive any security updates and are more susceptible to exploitation.

1.2 Research Questions

The primary issue we addressed in this research is whether we can enforce out-of-guest permissions to check access to the files of a system so that the attacker is not able to read or write critical files on the system. Following that we will address:

- What is the best way to implement a monitor for file access on the guest.
- What is the performance overhead.
- If this mechanism can be leveraged to identify a compromised system or a system actively being compromised.
- If it is manageable to monitor all files on a system or only specific ones.
- What is the best way to implement VMPCS-OGP on a guest and still provide usability and protection.
- If VMPCS-OGP can be used to discover how a system was compromised and attacker methods in compromising a system – sort of a honey pot approach.

- If we can return a valid error to the VM while denying access to a file, so that it does not reveal the extra security check imposed by the hypervisor.
- Can we enforce an append-only write policy for files like logs.

1.3 Organization

This paper is organized into five chapters. Chapter 1 introduces the concepts and thesis focus. Chapter 2 covers some background information for the platform used in this research, as well as some of the security solution already presented that make use of VMI. Chapter 3 analyzes the design and methodology of the implemented mechanism, and Chapter 4 discusses the performance testing results and presents our conclusions. Chapter 5 suggests future work.

CHAPTER 2:
Background

This chapter presents information about the relevant software and hardware. The first section gives a short introduction on virtualization and its benefits, and then describes the Xen hypervisor. The next section overviews the LibVMI application program interface (API) and DRAKVUF, the library and main application we will leverage, as well as the system call functionality and convention. Finally, we review some of the existing solutions that leverage introspection.

2.1 Virtualization

Running many and different services on a single operating system (OS) is an implementation method that vendors are abandoning, as mentioned in Rosenblum and Garfinkel [5]. In the past years advances in computing have enabled users to run a plethora of different software, which became a challenge to manage efficiently and securely because each service required a specific OS configuration. Over time hardware became inexpensive and service providers preferred to run one service per physical system to achieve higher security, since now each OS could be configured properly for the one service it was running. On the downside, running one service per physical machine resulted in the underutilization of hardware and capabilities, as well as increased maintenance costs. Hosting different virtual machine (VM)s on a single and powerful system (Fig. 2.1) solves many of the problems, as observed by Rosenblum and Garfinkel [5]. VMs cause resources to be used efficiently, with each service using only a part of the underlying hardware. VMs also allow easier security implementation, because it is much simpler to secure one VM running one service, than having to combine all of them into one. Additionally virtualization achieves redundancy between services, since each VM is independent from the rest, and any one failure does not affect the other VMs.

The advantages of virtualization do not stop there. Easy backup, restore, cloning, and migration of a system are just a few of them. Creating snapshots of entire machines and restoring to a previous state, in case of corruption or misconfiguration, has become a trivial task. Also, modern hypervisors implement a very solid and sophisticated VM isolation;

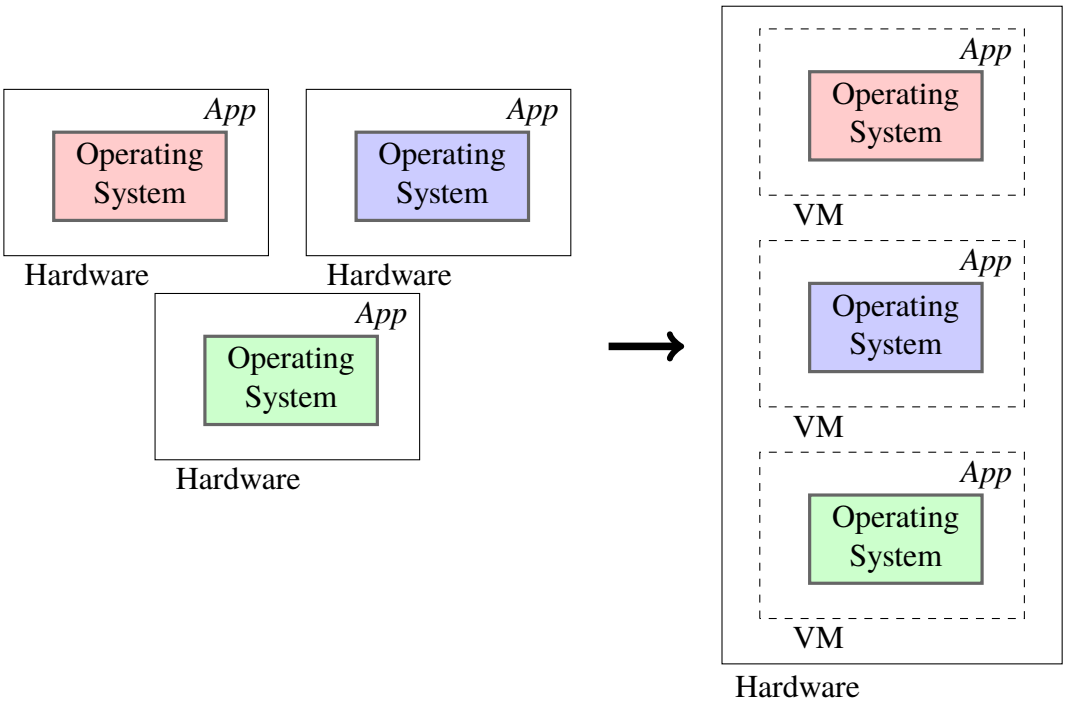

Figure 2.1. Evolution of software deployment from single OS to virtualization

pivoting from one VM to another has become extremely difficult.

Hypervisor is the software that drives this mechanism. It runs directly on the hardware, uses a separate OS installation, and resides outside all the guest VMs. At the same time, since the hypervisor manages the allocation and usage of all physical resources, it can see the internal state of each VM.

2.1.1 Hypervisor types

Different vendors provide their solutions in virtualization. Generally, hypervisors are separated in two categories: Type-I or bare-metal hypervisors and type-II or hosted hypervisors. Figure 2.2 shows the basic architectural difference between the two types.

Type-II hypervisors are applications which require a host OS to run on. Typical type-II solutions are VMWare Workstation and Oracle VirtualBox. These hypervisors work like any other application and the VMs run on top of them. Although they are simpler to manage for the average user, as well as for simple applications or use as testing environment, type-II hypervisors perform worse than type-I, as explained below.

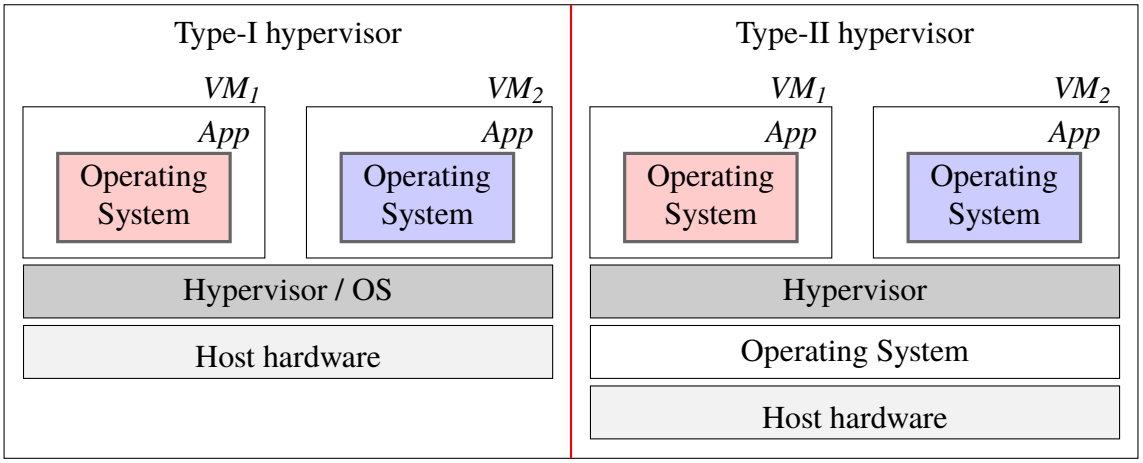

Figure 2.2. Architectural difference of type-I vs type-II hypervisors

Type-I hypervisors run directly on the hardware, managing the resources directly without the intervention of any host OS, providing a significant performance advantage. The advantage comes from eliminating the underlying OS of the type-II hypervisors. A Type-II hypervisor must ask the host OS for the resources the hypervisor needs to allocate every time, an action that produces performance overhead. Type-I hypervisors implement the resource management on their own since they run on a more privileged OS and they are actually part of it. The type-I hypervisors run and at the same privilege level with the OS and can manage the resources without asking the host OS. Therefore, type-I hypervisors provide a more efficient resource management of the hypervisor and its hosted VMs. Type-I hypervisors are most commonly used in server deployment and enterprise solutions, where performance and efficiency are important.

2.1.2 The Xen project

The product of the Xen project [6] is an open-source type-I hypervisor. Its small footprint and limited interface to the guest makes it more robust and secure. The hypervisor runs directly on top of the hardware, as depicted in Figure 2.3. It requires a host OS which acts as an interface between the hypervisor and the user, as well as paravirtualized guests. This host OS is called control or privileged domain, also known as Dom0, and runs at a more privileged level than the rest of the VMs. The rest of the VMs run on a lower privilege level and are called guest domains or DomUs.

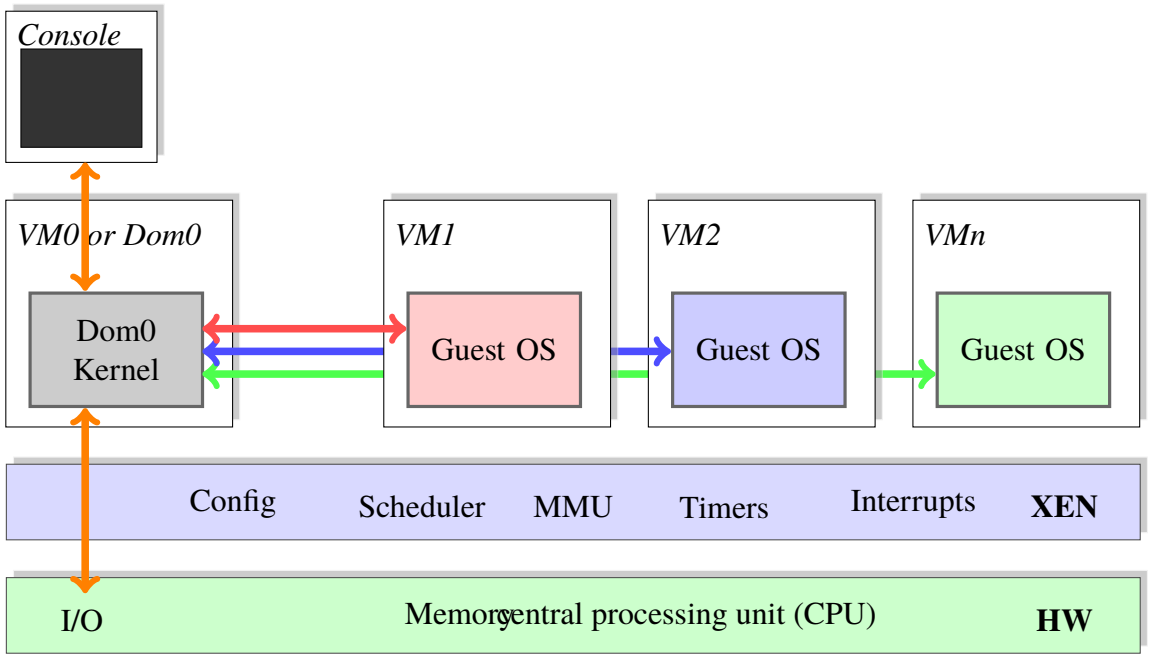

Figure 2.3. Xen Hypervisor Architecture

To understand how this happens, we need to introduce another CPU architectural feature, which provides different privilege levels for the execution of the CPU instructions, depending on the nature of the program invoking them. This mechanism called protection rings, is present on all modern CPUs and is used by all modern OSes. Protection rings are numbered 0 to 3, with 0 being the most privileged. Usually, applications run in ring 3, also called user mode, and the kernel and device drivers run in ring 0, also called privileged or supervisor mode. But, in order to allocate and manage the shared resources the hypervisor must run at a more privileged level than the guest OS, otherwise there is a conflict when the guest OS and the hypervisor try to manage a same resource. Initially, paravirtualization, a technique where OS vendors had to modify their kernels to run on a different privilege level, besides 0, like 1 or 2, was used to avoid that conflict between the guest OS kernel and the hypervisor.

For type-I hypervisors to work more efficiently and without any guest OS modification, CPU manufacturers have introduced a new ring -1 to support virtualization. The new ring, called hypervisor mode, is even more privileged than ring 0 and is employed only during hypervisor execution. This architecture is supported on newer CPUs that employ virtualization technology (VT), VT-x for Intel and AMD-V for AMD processors.

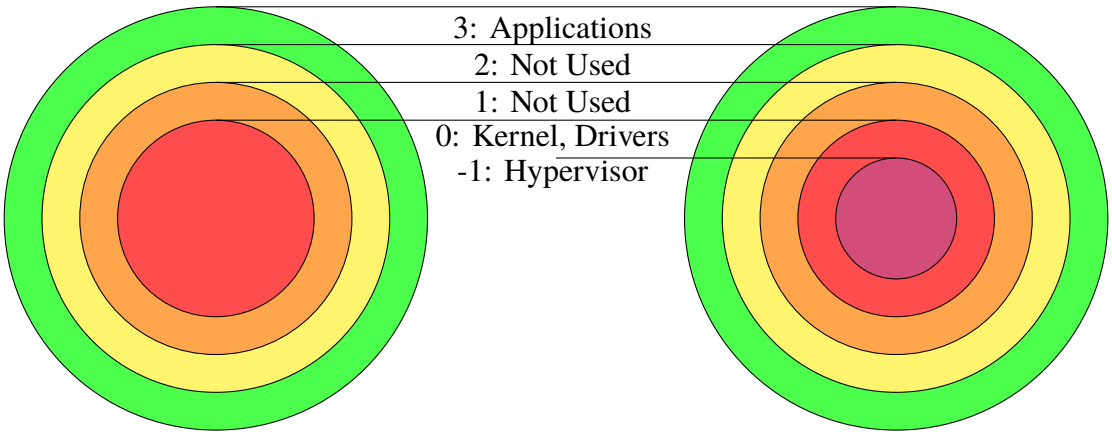

Figure 2.4. x86 protection rings

As virtualization keeps advancing, there is always the question of whether we can leverage virtualization to provide more than efficient sharing and usage of resources. The unique ability of the hypervisor to access the state of a VM, at the fine grain level of CPU registers and memory bytes, has been the center of research ever since the technology was invented.

2.2 Virtual Machine Introspection

In this section we introduce a rough timeline of the APIs, significant for our research, that were evolved around virtualization and virtual machine introspection (VMI). These include *LibVMI* and the later implemented *altp2m* for Xen. We mention *DRAKVUF* [4], a tool that employs LibVMI and altp2m to implement a virtualization based agentless black-box binary analysis system. DRAKVUF is the platform we based our solution on.

First introduced as a concept by Garfinkel et al [2], VMI leverages the more privileged status of the hypervisor to inspect the internal state of a VM. The Xen hypervisor was first to include introspection methods to inspect its guest VMs. Although these introspection methods where included in Xen, implementing introspection in a way that is secure and efficient was a non-trivial task. To make these methods more accessible to programmers, XenAccess [7] was implemented, as well as an API called *mem-events*. Because of strong research and security interest, introspection in Xen progressed and eventually LibVMI [3], a library that makes the implementation and automation of introspection on the Xen hypervisor easier, was introduced. LibVMI provides access to part of the hypervisors introspection methods to third-party applications, using a C or Python interface, the later called PyVMI.

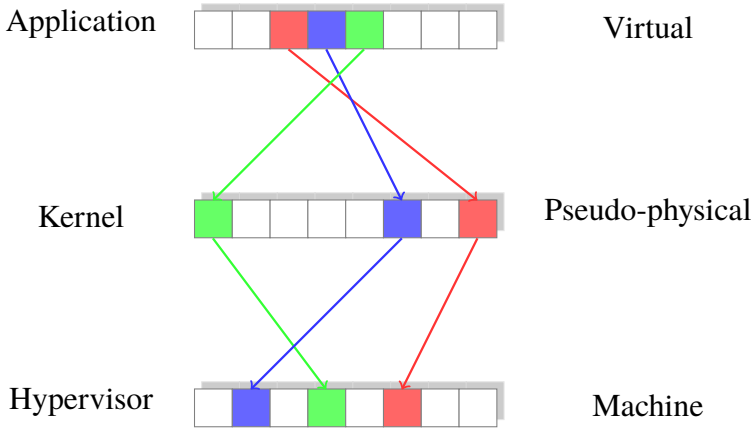

Figure 2.5. Hypervisor memory management concept

Initially, hypervisor memory management included an extra step in the memory access mechanism, because each VM assumes that it has complete control over the entire address space, and assumes that it writes directly on the hardware. Normally the OS would translate the virtual address used by an application to a physical address on the hardware. To a hypervisor, each VM is essentially an application. Since every OS will eventually try to write on the same physical address, the hypervisor must make a distinction between the VMs. To achieve that distinction the hypervisor assigns each VM a specific physical address space and tracks the overall memory usage with an additional page table (PT) translating between a VM specific guest machine frame number (GMFN) and the machine frame number (MFN), as explained in Chisnall [8].

With the introduction of input/output memory management unit (IOMMU), this extra step is no longer needed, because hardware extended page tables (EPTs) were included in the CPUs and the hypervisors can use these hardware EPTs instead of software ones, a method called hardware assisted paging (HAP(1)). HAP(1) implemented better isolation, and therefore enhanced security between the VMs, while at the same time the overhead reduced significantly. Following that development, as well as Intel's addition of 512 EPTs in its Haswell generation CPU, XenAccess and mem-events were redesigned and were evolved to a system called *altp2m*. One of the most critical changes that came with altp2m was the concurrent assignment of multiple EPTs per VM (Fig. 2.6). Additionally, monitoring processes of multi-virtualCPU guests is more secure, because each virtual CPU can be assigned its own EPT. This was a significant improvement, as the hypervisor can keep track

of different EPTs with different permissions, which can change during the execution of the VM. Other solutions keep only one EPT per VM, resulting in a less secure and isolated virtual environment between the VMs and the VMs' processes.

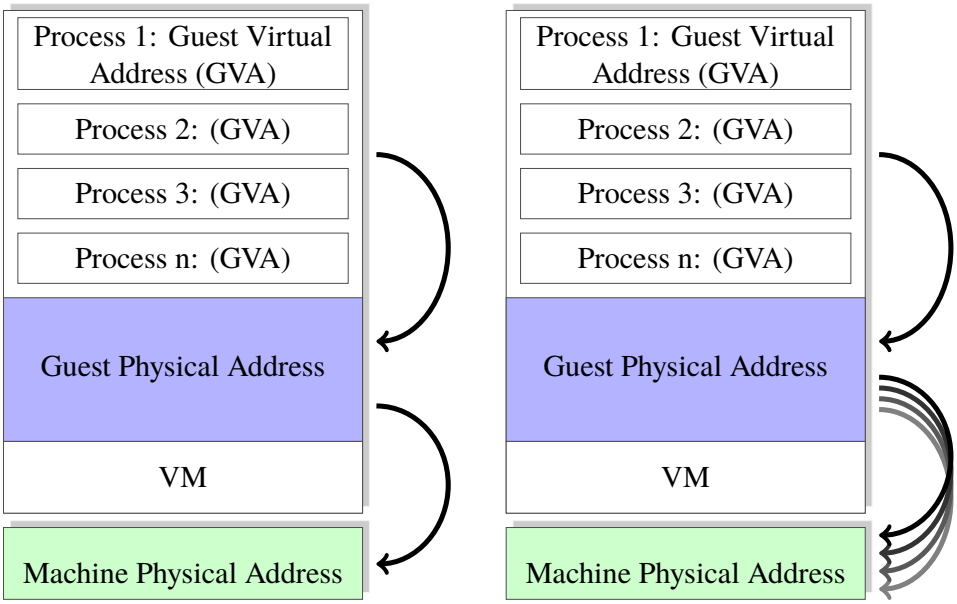

Figure 2.6. Normal vs altp2m multiple EPT assignment

LibVMI, as mentioned earlier, is an API which provides exposure to a subset of Xen's VMI functionalities, as well as other platforms. LibVMI makes it possible to monitor the state of any VM, including memory and CPU state. Memory can be accessed directly, using physical addresses, or indirectly with the use of virtual addresses, OS symbols, and user application symbols. It can monitor memory and register events, like memory read, memory write, register value change, and provide notifications for them, allowing this way the execution of callback functions, while the monitoring application resides outside the VMs and accesses the VMs through the hypervisor (Fig. 2.7).

LibVMI focuses in a subset of introspection methods that provide memory reading and writing capabilities from running VMs. It also provides methods for accessing and modifying CPU registers, as well as helper methods to pause and unpause a VM. Accessing a VM's memory space is not a trivial task. After detecting where the page directory is, a scan of the page tables follows to detect the memory mapping of the running process. This gets translated to a virtual address, which later, the hypervisor translates to a physical address.

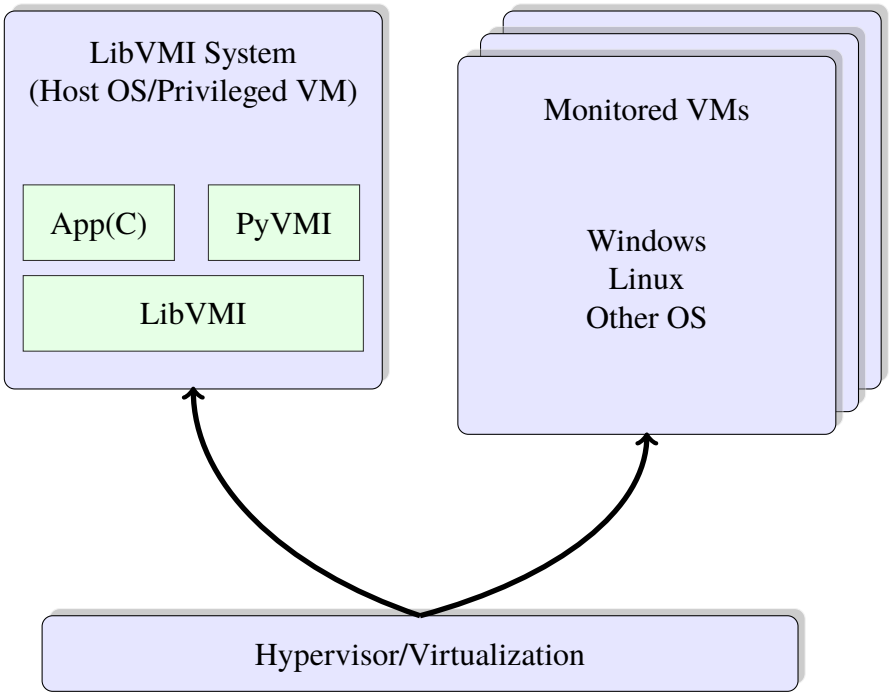

Figure 2.7. LibVMI out of guest access of VM state

Figure 2.8 shows a slightly different request, that of reading a kernel symbol.

Xen's introspection methods significantly impact system security. The monitoring application resides on the host and accesses the VMs' state from the hypervisor, which implies a zero-footprint monitoring tool from the VM's perspective. The monitor does not leave a trace of its action that can be detected from inside the guest.

Although this development was game-changing, it had its drawbacks. Just monitoring that values of specific parts of memory, or the CPU registers, over a time interval to make any inferences about the running state of the VM leaves the VM vulnerable during the waiting period. A solution is to trap the memory regions that we want to monitor for access or modification. But this can be detected by a knowledgeable adversary.

To solve this problem, LibVMI, altp2m, along with the substantial number of EPTs on the latest CPUs, were were combined in DRAKVUF [4], a dynamic malware analysis platform. One of DRAKVUF's most significant key features is that it traps the memory addresses the user wants to monitor for access. When the event gets triggered, the EPT

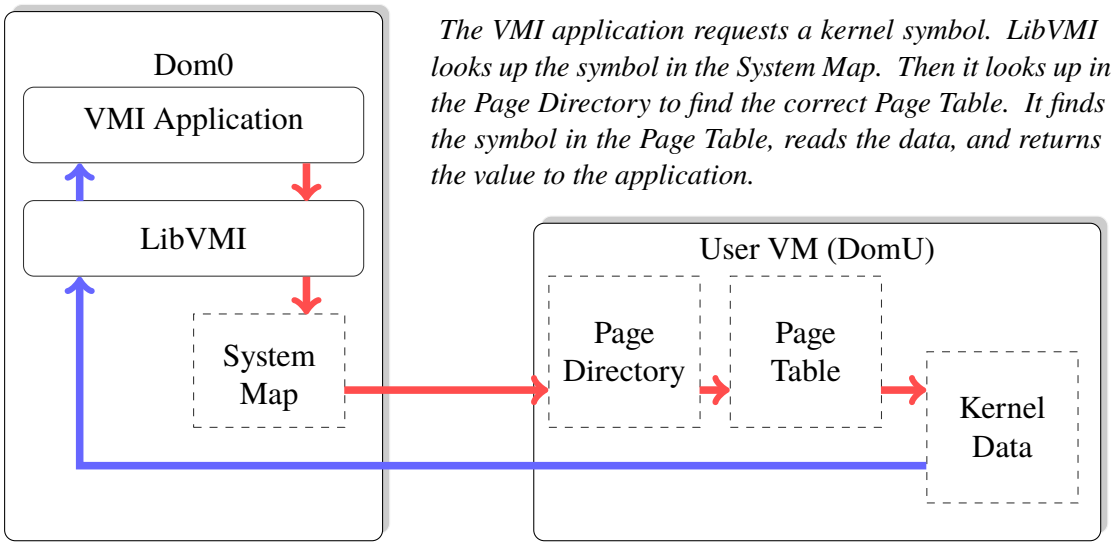

Figure 2.8. Using LibVMI to access the value of a kernel symbol

with the trapped address gets swapped with the original, so that the execution of the guest VM continues. This allows the monitoring of an arbitrary number of memory addresses, providing notification and response capabilities on every such event, while at the same time being untraceable from inside the guest.

2.3 System Calls

Modern OSs are responsible for allocating their resources efficiently and securely to themselves, as well as to the user level applications. The part of the OS assigned to manage these resources, like memory, hard disk drive access, or CPU time, is the kernel of the OS. The kernel, which runs in its own space, is the heart of the OS that makes everything work in harmony without conflicts or resolves them if there are any. When an application is running, it runs in the so-called user-space. This distinction exists to prevent applications from having direct access to the underlying hardware and is enforced with the protection rings, explained previously. The running application has no knowledge of any other application being executed on the same machine, and whenever it requires some resource, it asks the OS through the kernel. The kernel, on its behalf, accesses the hard disk drive, allocates memory, or executes other commands that are considered privileged and the application cannot execute. It handles all the low-level details of what the application asked and returns the results of the action.

This very complicated software is the most crucial part of the OS. Therefore, not every process can access the kernel directly or invoke all the kernel's functions in order to avoid corruption or misuse the low-level access the kernel has, to gain access where a process should not. This limited interface to the kernel, a sort of protection mechanism, is called a system call. The details of making a system call depend on the OS.

Programming with a high-level language usually does not involve making system calls directly. Most languages have implemented wrappers for making a system call and simplifying the system call interface. Regardless, the application will eventually have to make a system call to access some of the system's resources. Files is one type of resource an application needs to request access from the OS. This is performed with the `open()` system call. Access to input devices, like a keyboard, is also requested from the OS with the use of the `read()` system call.

2.4 Related Work

Whether resulting from user error, or targeted malicious activity, system compromise is bound to happen because of errors in the running programs. This eventuality led researchers to invest their resources in VM security.The Introspection concept gave birth to numerous interesting solutions, that target a more critical issue of the information world, that of computer security. Some solutions focus on the analysis part, where by leveraging the hypervisor's introspection methods gain better insight and understanding of the behavior and impact of a malware, so that it can be successfully intercepted. Other solutions take a more active role by trying to protect crucial parts of a running VM. They prevent the kernel from becoming corrupted, or provide secure access to parts of memory where critical information or applications are stored. These solutions can provide valuable information on which events and actions led to a compromised system, or protect the vital OS space from being corrupted by malicious activity, each of them in its own unique way. The following categories of methods of securing VM-based systems represent only some of the solutions produced so far, and the categories were based on the work by Bauman et al [9].

2.4.1 In-VM Monitoring

In-VM solutions implement part of the functionality inside the VM. They employ an inside agent to gather information on the VM execution state and use the elevated privileges of the hypervisor to protect the agent from corruption or subversion. Depending on the application, we can further refine the classification in terms of detection, prevention. There are also some recovery solutions, but we consider them outside the scope of this research. Working in a VM to gather information for the hypervisor can become a very intensive task, increasing the performance overhead. The hypervisor, as well as every VM, is a complete OS, running its processes, applications, and scheduler and intercepting its own interrupts. There is additional performance overhead when the execution switches between a VM and the hypervisor and vice versa. The pair of events related to hypervisor and VM switching are called `VM-exit` and `VM-entry` (figure 2.9). Having a monitoring and logging application on the hypervisor triggers a considerable number of VM-exit events. This is a problem some of the following solutions tried to address by using different approaches.

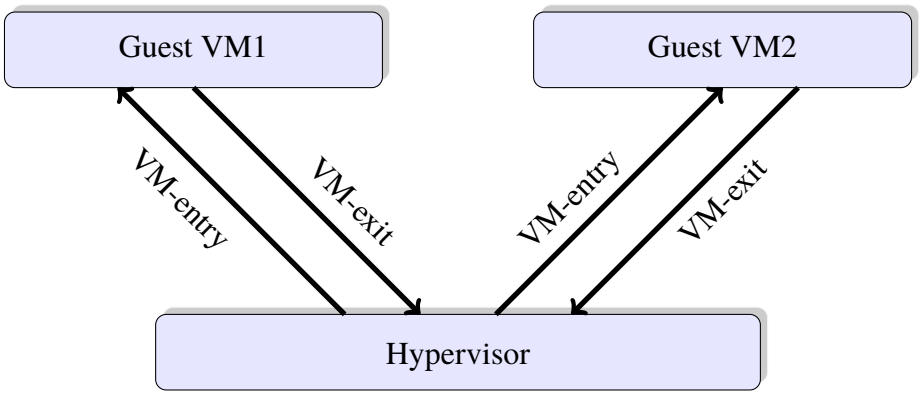

Figure 2.9. VM-exit and VM-entry events

Detection

To prevent this overhead, a monitoring solution, SIM [10], used the hypervisor in the following way. The hypervisor, since it provides all the resource allocation, can mark the memory pages allocated to a VM different, than the guest OS would. It can mark a page read-only when the OS marks it as read/write. This will trigger a VM-exit event and the hypervisor can act according to a different policy than that of the VM's OS. So SIM is placed inside the VM, monitoring the guest OS, but at the same time it is protected by the hypervisor by being placed on a protected region of the VM's address space.

Gathering information at the hypervisor level though, presents a new problem. Each action collected can have been executed by different processes. This uncertainty makes it harder to understand the higher level action being executed. It is a semantic gap between the hypervisor and the guest VMs. Virtuoso [11] is a tool that tries to bridge that semantic gap by automating the process of extracting OS kernel information relevant to introspection. It runs a helper program inside the VM, which yields the wanted result. It analyzes the execution trace of that helper program and generates the introspection code that will give the same result when executed from the hypervisor. This method helps gain some knowledge about the internal machine state without having the required intricate knowledge of OS, but from the hypervisor's point of view.

Prevention

Lares [12], in the same manner, tries to modify the guest OS minimally, so that the code used for monitoring can be protected easily, while all the introspection and decision making code is placed in a security VM. The two communicate through the hypervisor, which protects the hooked code in the untrusted VM, while at the same time provides information to the security VM. It also provides communication between the VMs, so that the decision making on the security VM can be enforced upon the untrusted one. In this case, the monitoring happens on process creation, allowing or denying the execution of programs, as defined in a white-list.

SHype [13] is a modified hypervisor that implements mandatory access control (MAC) on shared resources between VMs. SHype is used also in [14], to provide a more fine-grained MAC on data flow between VMs and services. Hyperlink [15] implements a hybrid of protected in-VM monitoring alongside MAC-based hypervisor protection, for guest VM and hypervisor protection.

InkTag [16] introduces many different new concepts to run high assurance process (HAP(2)) in an untrusted OS. The threat model for this approach is more advanced and sophisticated. Inktag, to protect the HAP(2), employs many different mechanisms, on various levels, to ensure that there is no data leak or malicious intervention during the HAP(2)s runtime.

Inktag also introduces paraverification [16], where the kernel is required to perform some extra tasks, to provide the hypervisor high-level information about the process state. This

way, the hypervisor can easily determine the high-level effects of low-level actions. Furthermore, the HAP(2) does not interact directly with the kernel. This is done by an untrusted trampoline code, which is responsible for making the system calls instead of the HAP(2), and receiving the system call results from the OS, and, after validating them, return them to the HAP(2).

To protect the contents of the HAP(2)s' memory address space, InkTag employs two EPTs: one for use during untrusted execution, which is visible by the untrusted OS, and one for use during trusted execution which is visible and used only by the hypervisor. In addition, if a page from the HAP(2)'s address space needs to be evicted, InkTag hashes the contents and encrypts them before they get written on the disk. This way, it provides protection against malicious modification and access. Also, to further protect the HAP(2) and its files, a different access control mechanism is used. Each process and file is followed by attributes, which are used to enforce an access policy, such that it will protect the files, the processes, and their spawned processes. InkTag also uses a different convention to address memory and files, with the use of object identifiers (OI), an internal representation visible and known only to the HAP(2) and the hypervisor. These are used to define the permissions each HAP(2) has. Finally, InkTag modifies the actual media layout, to inject file metadata, which are used to provide crash consistency. These metadata are not visible by the untrusted OS, since these sectors are not included in the media view of the OS.

Although InkTag provides many assurances for the secure execution of a HAP(2), the need to recompile applications so that they can run securely, poses a significant drawback and compromise of usability.

Using a similar approach, Overshadow [17] provides a one-to-many memory mapping from the VM to physical memory, as well as other mechanisms to further protect the applications and their data. The actual data in memory depend on the process trying to access them. The contents get encrypted and hashed for untrusted processes and decrypted when the trusted application tries to access them.

To manage secure application execution inside a compromised OS, Haven [18] takes a different approach. To protect the application, Haven employs Intel's software guard extensions (SGX). SGX allow a process to define a secure region of address space, called enclave. Haven puts the whole application in an enclave and uses an in-enclave library OS

for the interactions with the OS.

On the downside, InkTag and Haven were attacked in [19] with the use of controlled-channel attacks, resulting to the extraction of substantial amounts of sensitive information from protected applications. Complete text documents were extracted, as well as outlines of JPEG images, showing that data protection during process is not a trivial task.

2.4.2 Out-VM Monitoring

Having a monitoring tool completely on the hypervisor has its benefits, but also a significant drawback. Although everything is visible from the hypervisor's perspective, the data collected miss context. It is extremely difficult to understand the context by analyzing memory and CPU register values during every execution cycle, a semantic gap that needs to be filled. This section will present some of the out-VM solutions. Some work on raw collected data, while others try to bridge the semantic gap to better understand the high-level commands being executed in the VM.

Detection

ReVirt [20] is a logging application. By using the hypervisor's VM access, it creates extensive logs of a VM's execution. Since the hypervisor has unlimited access to the state of the VM, ReVirt can collect and record enough information to be able to recreate and simulate the execution of the target machine. This can be very valuable for collecting malware activity data even after the system has been compromised, hijacked, or even replaced. The replay data can prove very useful in the malware analysis field, as every non-deterministic action of a malware is recorded and deterministic results can be recreated, providing a full view of the system and the malware's impact at every step of the malicious activity.

From the moment the VM starts booting, Macko et al [21] uses the ability of the hypervisor to transparently access the running VM's internal state to collect system-level provenance.

Using a different approach, Crawford and Peterson [22] implements a mechanism to detect insider threats. It uses VMI to stealthily monitor the user's actions and detect suspicious activity that correlates to an insider threat. Although this alert mechanism is very useful, especially due to its transparency, the attacker still gets access to the information he wants.

When the introspection idea was conceived in Garfinkel et al [2], it was utilized to create a hybrid intrusion detection system (IDS). The IDS solution gained the best of both worlds, host intrusion detection system (HIDS) and network intrusion detection system (NIDS) by being placed on the hypervisor. Since it is placed outside the VM, it has the advantage of not being prone to detection, attack and corruption, or evasion. It can directly monitor the network traffic, given that the network interface card (NIC) is a common shared resource. On the other hand, by having the hypervisor's introspection capability, it can act also as a HIDS by monitoring the actual system behavior and execution.

Other solutions have been proposed to fill the semantic gap between the hypervisor and the guest VM like Strider Ghostbuster [23], PoKeR [24] and VMWatcher [25]. All of them employ different techniques, but unfortunately, as later researchers, like Mahapatra and Selvakumar [26], mention, they fail at a point because this semantic gap is difficult to bridge.

Prevention

This semantic gap was also addressed in [27] with a technique called process out-grafting. Instead of monitoring the VM as a whole, this method focuses on each separate process, for a more fine-grained execution monitoring. This is done by implementing two new techniques. The first is called on-demand grafting, which can relocate a running process from the guest target VM to a security VM. This effectively bridges the semantic gap, as for all intents and purposes the process is running on the same system as the monitor. This way, the monitor can intercept all instructions executed by the suspicious process without the need of hypervisor intervention. The second technique, called split execution, makes a logical separation on the execution of instructions. If the process runs in user-space, it continues to run on the security VM. When there is a kernel request, like a system-call, it executes that instruction on the target VM. That technique isolates the monitor from the suspicious process, since they do not run on the same kernel, while at the same time from the suspect's process perspective, it is still running inside the target VM.

Furthermore, SecVisor [28] and HUKO [29] propose a kernel integrity method that protects the kernel against rootkit code injection. In this case, SecVisor and HUKO are part of the hypervisor. They permit user allowed code execution, while at the same time preventing malicious code execution.

Sentry [30] does a more granular kernel protection by preventing low-trust kernel components from altering security-critical data used by the kernel to manage the system and itself. It protects dynamically allocated memory, is isolated from the untrusted kernel by running on the Hypervisor, and reduces the overhead by monitoring only the kernel related memory pages for suspicious activity.

Paladin [31] first introduces the concept of Out-of-Guest access control list (ACL). The ACLs in this case have a rule-based system that marks whole folders or files with generic permissions. By generic permissions we mean that they affect all users and groups the same. There is no user-specific or group-specific ruling. All file accesses fall in the same category. Therefore, Paladin provides its file security at a more coarse level than the one desired. Additionally, it uses an in-guest module, which although relatively small and protected, still runs in the guest VM. Also, as stated in [31], this ACL implementation introduces many usability problems, like an upgrade on the running applications. To perform such an action the system must be taken offline, its ACL rules must be changed to perform the upgrade, and restore the system online.

A more direct approach to file integrity is presented in [32]. Nasab tries to protect the OS from accessing maliciously modified files. The target VM is deployed offline and all the files are signed digitally using a private key. The digests are stored on the hypervisor. When the process has been completed for all the files to be protected, the VM gets online. During its execution, whenever a file is accessed and before it gets loaded into memory, the system retrieves its digest and compares it to the copy on the hypervisor. If the file has not changed, access or execution continues, otherwise access or execution is denied.

Table 2.1 shows a representation of the key features of the solutions presented above.

As far as we know, all work on VM monitoring and security focuses on kernel and OS protection, malicious activity monitoring, extensive logging for replay and online or offline forensic purposes, or secure resource sharing among VMs. Only Paladin [31] provides a protection for the actual files of the system. But even this solution, as mentioned before, provides a generic out-VM ACL approach, and can introduce various usability issues. Also, it will need further development to protect against newly created attacks, which can be maliciously access files when the VM has been compromised. Furthermore, it is a in-VM solution, as there is code of Paladin running in the guestVM, leaving this way

Table 2.1. Overview of solutions

Solution	In-VM	Out-VM	OS		File Protection	
			Detection	Prevention	Detection	Prevention
SIM [10]	✓	-	✓	-	-	-
Virtuoso [11]	✓	-	✓	-	-	-
Lares [12]	✓	-	-	✓	-	-
SHype [13]	✓	-	-	✓	-	-
InkTag [16]	✓	-	-	✓	-	-
Overshadow [17]	✓	-	-	✓	-	-
Haven [18]	✓	-	-	✓	-	-
ReVirt [20]	-	✓	✓	-	-	-
Macko et al [21]	-	✓	✓	-	-	-
Crawford et al [22]	-	✓	✓	-	-	-
VMI [2]	-	✓	✓	-	-	-
Strider Ghostbuster [23]	-	✓	✓	-	-	-
PoKeR [24]	-	✓	✓	-	-	-
VMWatcher [25]	-	✓	✓	-	-	-
Srinivasan et al [27]	-	✓	-	✓	-	-
SecVisor [28]	-	✓	-	✓	-	-
HUKO [29]	-	✓	-	✓	-	-
Sentry [30]	-	✓	-	✓	-	-
Nasab [32]	-	✓	-	✓	✓	-
Paladin [31]	✓	-	-	✓	✓	✓

a footprint of its presence. VM security is a very active research field, producing many solutions, each with different focus, but generally surrounding the malware protection realm, as depicted in table 2.1 and even more extensively in Bauman et al [9], an extensive survey on hypervisor-based solutions.

Because zero-day vulnerabilities are constantly discovered and exploited, we do not want to focus in detecting specific behavior to assess the presence of malware and restrain it, since these behaviors change and evolve with the development of new attacks. We want to focus in a more generic case where we expect the running system to be hacked. We want to protect files in a compromised running VM against insider threats, rootkits, and malware in general.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Design and Implementation of Ferify

In this chapter we will discuss the design, including the threat model and assumptions made, and the implementation choices that we made for this research.

3.1 Overview

In this research, we extend the Xen hypervisor by leveraging its virtual machine introspection (VMI) capabilities to create a system which we call `ferify` that protects some critical files on a virtual machine (VM)'s mounted filesystem by intercepting and monitoring *all* systems calls that may operate on these files. `ferify` maintains its own file access control lists (ACLs) and allows a system call to read or write a file *if and only if* the ACL permits the action explicitly for the userid (UID) and groupid (GID) combination of the calling process. We call this ACL the shadow access control list (SACL), to differentiate it from other ACLs configured on guest VMs. It is important to note that the content of SACL cannot be read or modified by *any* process including kernel processes from a guest VM. The above permissions can be different compared to those on the guest VM, allowing for the enforcement of different file access policy than the ones enforced by the guest VM, even while under VM compromised attacks. The ACL policy enforced by the guest operating system (OS) remains active, meaning that if for some reason the SACL entry allows file access but the guest OS ACL does not, the file will not be accessed.

Before presenting the design and implementation details, we first briefly discuss why our solution is secure. We chose to monitor which files are being accessed by trapping on the system calls that are being invoked.

As mentioned in [33], a User Mode process cannot access directly the hardware. Each file operation must be performed in Kernel Mode. The OS will not allow direct access to hardware. If a process needs file access, it needs to perform a system call, asking this way the OS for the desired operation. The OS checks if the request is valid and if it is the OS accesses the file on behalf of the process and returns to the process the result of the operations, as shown in 3.1. This requirement, to make a system call for *all* file operations,

provides the place in the OS executable code, which if we monitor we can extract the information of all files being accessed. So, we cannot miss *any file any process* will try to open.

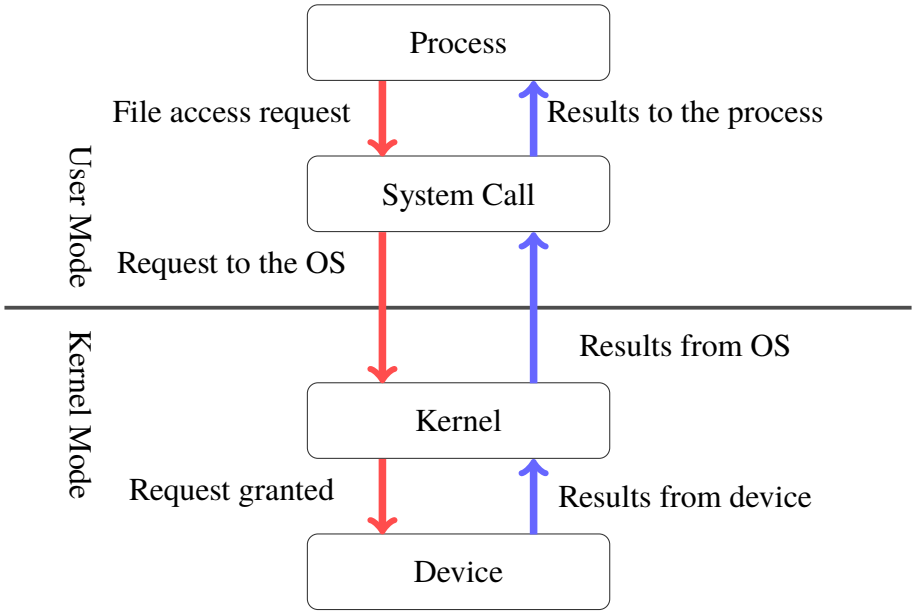

Figure 3.1. System call information flow

Table 12.1 in [33] shows the list of system calls relevant to filesystems. According to that table, the system calls of importance to us are `open()`, `rename()`, `link()`, `symlink()`, `unlink()`, and `truncate()`. Some of these system calls have been evolved over time to provide better security and to eliminate problems. Such an evolved system call is `openat()`. Going through the available system call list in `/usr/include/asm/unistd_64.h`, we assess that the system calls mentioned in table 3.1, where number is the system call number as it is passed to the kernel, provide *full coverage* of the possible ways a file can be accessed when being on a mounted device.

As explained in [34], it is possible for an attacker to modify the credentials of a process after acquiring `root` privileges. This exploit is a significant security issue for `ferify`, since it collects the UID and GID of the running process from the VM's kernel memory. To avoid the success of such an exploit, `ferify` monitors the credentials of *all* running processes inside the guest VM, and additionally monitors the creation of new processes to record their credentials as well. Then, according to preset rules `ferify` decides whether a change in

the credentials of a process is allowed or not, and acts accordingly. This way we secure the validity of the information we collect from the guest VM.

Table 3.1. Trapped system calls related to file operations

System call	Number	System call	Number
`open()`	2	`openat()`	257
`name_to_handle_at()`	303	`open_by_handle_at()`	304
`rename()`	82	`renameat()`	264
`renameat2()`	316	`truncate()`	76
`link()`	86	`linkat()`	265
`symlink()`	88	`symlinkat()`	266
`unlink()`	87	`unlinkat()`	263
`execve()`	59	`execveat()`	322

LibVMI, and therefore DRAKVUF, needs for the guest VM to have reached a state, after launching, to be able to start monitoring the guest VM. After we launch a VM there might be a small time window when the guest VM is running without the protection of our system. Until we find a way to find the exact time LibVMI can be initialized to protect the guest VM, we will consider attacks that modify the running system during boot-time outside the scope of this research. To prevent the system from being attacked during that small time-frame, we do not connect the VM to the network until this initialization process is completed.

3.2 Threat Model

Computer security has been evolving because the attackers methods evolve, too. Modern OSes and applications are so complex that they introduce many bugs in their code. Some of these bugs are benign, but some are serious enough to allow security breaches like remote access to a system, administrator/root access, and arbitrary code execution.

For this research, we have adopted a *moderate* threat model where we assume that the guest VM is insecure. We assume that physical access to the hosting machine is restricted and the attacker cannot use physical media like USB sticks or CD-ROMs to compromise the host system. The protected files are only remotely accessible and protected by a public-key authentication and encryption scheme (i.e.,*SSH*). The private keys of authorized users are secure, while the attacker may have obtained *root* privileges on the VM through a successful attack. Assuming that the target VM will be accessible only remotely, reflects

many applications and systems working over a network connection as well as cloud solutions. A significant concern in computer security is that of replay attacks. Even if a legitimate user connects using *SSH*, there are attacks that can monitor network traffic on the host and steal sensitive data after they are decrypted. We consider this type of attacks and how to mitigate them orthogonal to this research, as the *SSH* protocol is widely used and evolving.Finally, The files we want to protect are on a mounted disk.

Moreover, we assume that the OS installed on the guest VM is not trusted. This essentially means that the adversaries can gain root privileges, allowing them this way to modify system executables, as well as load kernel modules during runtime. This allows for kernel memory modification.

We consider the hypervisor along with its Dom0 to be secure and trusted; hypervisor vectored attacks or how to protect the hypervisor is outside the scope of this research.

3.3 Requirements

The goal of this research is to provide a virtualization extension that will extend the granular-level file access control of the Linux OS. The requirements we defined for our system are:

- **(R1)** The solution must be *out-VM*, to avoid modification from the potential adversary.
- **(R2)** The system must remain *efficient and usable*, by not introducing significant overhead on the runtime of the VM, as well as by not enforcing many restrictions to the users.
- **(R3)** *All* relevant system calls must be monitored.

3.4 Design

We used a type-I hypervisor instead of type-II, as type-I hypervisors are more efficient and deployed more as a commercial solution, while type-II are used mostly for testing and analysis. We chose the Xen hypervisor as it is open-source and is used widely, so the results of the research can be used in a variety of applications. By leveraging Xen's introspection methods, we created an *Out-VM* monitoring agent, which runs on Dom0, completely *outside* the VM, conforming this way with **(R1)**. Also, by ensuring that there is no code running on the guest OS, we increased the deployment speed, as there is no need to modify the guest

VM in any way. The required pre-deployment configuration for the guest VM is kept to a minimum and does not involve any modification, only information gathering.

As a platform to base our solution, we chose DRAKVUF [4]. DRAKVUF provided us a stealthy monitoring base, as it leverages alternate extended page table (EPT)s with different permissions, preventing any detection from applications inside a VM. We had to restrict some of the usability of the system, although not during normal execution, to achieve the file *confidentiality*, *integrity* and *availability* we wanted, as explained later in this chapter. Therefore we assumed that **(R2)** is achieved in the part of restrictions, although a few restrictions apply, mostly concerning the `root` user. Overhead and usability of the system will be discussed in chapter 4.

We employed the stealthy property of DRAKVUF to make the process of file protection *completely transparent* to the guest OS, retaining a *zero-footprint* monitor on the guest. DRAKVUF also helped bridge the semantic gap between the hypervisor and the VM with the use of a Rekall profile [35], by having access to selected kernel structures. Furthermore, we wanted to employ a per-user ACL, enforced on specific files or whole folders, which are sometimes not essential to the OS, but *essential to the user*. Generally, we wanted to protect any type of data, regardless of the content. We improved *confidentiality* by denying even read access, *integrity* by denying write, and *availability* by protecting deletion or moving of the files. This mechanism must also extend to the `root` user, since our threat model assumed that the system is compromised. To achieve that, we intercepted *all* relevant system calls from *all* users and verify the validity of the request.

Monitoring the execution of the system calls in table 3.1 and validating the request made to the guest OS kernel is sufficient for ensuring the guest VM file confidentiality, integrity and availability, as we want to enforce it, thus conforming to **(R3)**.

3.5 Implementation

Following we expand on how we implemented the mechanism to provide the file access security of our system.

3.5.1 Architecture

`ferify` was developed as a plugin for DRAKVUF. As such, it is a C++ class that extends the class `plugin`, as per requirements of DRAKVUF. It is located in the directory `src/plugins/ferify/` and consists of two files, `ferify.h` and `ferify.cpp`, as show in figure 3.2.

Figure 3.2. `ferify` directory tree

3.5.2 Shadow Access Control List

During the initialization of our DRAKVUF plugin, we read the SACLs we had created for the VM to be protected. The SACL is implemented in the form of four hash-tables to improve search speed. Table 3.2 depicts the structure created for storing each SACL entry.

Variable Name	Variable type
`pathname`	`char *`
`mode`	`unsigned int`
`u`	`uid_t`
`g`	`gid_t`

Table 3.2. `struct protected_files` memory layout

The above memory structure is used for both protected folders and protected files. Moreover, we create two of each; one for the `root` user and one for the rest. Searching in the SACLs is done with the pathname of the file being accessed as the hash-table key. The return value is a pointer to the structure of table 3.2.

In the SACL file permissions are set according to the Linux permission bits schema. This means that the last 3 digits of the *mode* field, when encoded in octal form, define the permissions we want to enforce. The first one defines the owner permissions, the second the group permissions, and the third the other user permissions. The number itself is the sum of the permissions 4 for read, 2 for write, and 1 for execute. As an example, if we encounter or set permissions 744, it means that the owner can read, write and execute the file, while anyone in the same group and everyone else can only read the file.

The SACL's format was kept as simple as possible so that editing and reviewing it is easy. Additionally, by keeping the format of the Linux ACL, we provide to the system administrator a familiar permissions schema, so it is easier to understand and manage. Figure 3.3 shows an example, which we will analyze later.

Pathname	Permissions	User	Group
`/home/user/Documents/readme.txt`	`100644`	`1000`	`1000`
`/home/user/Desktop/credit.pdf`	`100400`	`1000`	`1000`
`/home/user/Documents`	`140220`	`0`	`0`

Figure 3.3. SACL sample

The system keeps four SACLs: two for all non-root users and two for root, since this account is of greater significance. Furthermore, two different checks are being performed. First it checks for a protected folder, as it is a more generic case. If no entry is found, it then checks for specific files in the list to match.

We chose to create two separate SACLs to improve the permissions policy of our solution. By making this separation we allow for the creation of separate permissions for normal users and the `root` user. It makes managing of different permissions easier. Besides this coarse refinement of the SACL entries, the level of protection applied to the files is the same, whether applied to the root user or any other user.

As we see in figure 3.4, the SACL for protecting files from the `root` user is more simple. Since it is targeted for this specific user, we do not need the entries for owner and group. Furthermore, the permission bits for group and others are ignored when parsed, since they are meaningless.

Pathname	Permissions
`/etc/shadow`	`100400`
`/etc/pam.d/su`	`100000`

Figure 3.4. root user SACL sample

Before moving on, we must emphasize that the system does not alter basic properties of the files that are being protected. It does not change the owner or the group, since this requires intervention in the VM. Although in the SACL we can define a different owner, the generic effects is denial of access. This means that we cannot change who can access a file, rather

we can change who cannot. This system acts as a supplementary and more fine-grained access control mechanism to make more strict file access policies. Therefore, if we change the owner of a file in the SACL, we essentially prohibit access to that file by the owner; we do not specify a new one, as the final call for file access comes from the unmodified guest OS.

3.5.3 System Call Interception and Admission Control

All applications running in user-space need to ask the kernel to access a file. Applications do not have knowledge of the low-level OS and device details to access the files they need, so they request from the kernel to do that work for them. The kernel accesses the requested file using the device drivers and, when the operation is completed, returns to the application a handle to that file, called file descriptor. This happens for many operations restricted to the kernel for security reasons. Also, it provides an abstraction to the applications, which are written without the need of the knowledge of device specifics and work on variations of the underlying hardware running the same OS.

For applications to be compatible to OS version upgrades and portable between different systems, a specific standard calling convention of these kernel functions is needed. This calling convention is a system call. System calls are specific entry points to the kernel, which, when provided specific arguments perform an operation on behalf of the application. Many system calls exist, each performing a different operation. We will focus on those who are relevant to accessing files, whether to read or modify. These are depicted in table 3.1.

To gain the insight on what files are being accessed, we need to know whenever this event happens. We will use DRAKVUF to create a trap on all the afore-mentioned system calls.

This gives us the opportunity to stop the VM execution when these system calls are called. At this point, we access the registers related with each system call to retrieve the information we need to perform the validation of the requested call. Figure 3.5 gives an overview of the flow of information during a trapped system call.

The arguments to the system calls for the 64-bit Linux OS we used as our test platform are passed to the kernel through the registers in the order of `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`, while the system call number is passed in `rax`. Table 3.3 shows which arguments need to

be passed to each system call on each register for it to perform the requested operation.

Table 3.3. Arguments of file related trapped system calls

Syscall Name	rax	rdi	rsi	rdx	r10	r8
open	2	const char *pathname	int flags	int mode		
openat	257	int dirfd	const char *pathname	int flags	int mode	
name_to_handle_at	303	int dirfd	const char *pathname	struct file_handle *handle	int mount_id	int flags
open_by_handle_at	304	int mountfd	struct file_handle *handle	int flags		
rename	82	const char *oldpath	const char *newpath			
renameat	264	int olddirfd	const char *oldpath	int newdirfd	const char *newpath	
renameat2	316	int olddirfd	const char *oldpath	int newdirfd	const char *newpath	int flags
link	86	const char *oldpath	const char *newpath			
linkat	265	int olddirfd	const char *oldpath	int newdirfd	const char *newpath	int flags
symlink	88	const char *oldpath	const char *newpath			
symlinkat	266	const char *oldpath	int newdirfd	const char *newpath		
unlink	87	const char *pathname				
unlinkat	263	int dirfd	const char *pathname	int flag		
truncate	76	const char *pathname	off_t length	int flag		
execve	59	const char *filename	char *const argv[]	char *const envp[]		
execveat	322	int dirfd	const char *pathname	char *const argv[]	char *const envp[]	int flags

3.5.4 System Call Hooking

To achieve the above mentioned system call intercept we need to place traps to the system calls of interest. This gets implemented by DRAKVUF. LibVMI reads the Rekall profile of the guest VM to get the base address of the kernel symbol table. DRAKVUF then

starts from that base address and searches for the system call table. This table includes the function pointers for all supported system calls. Going through that table makes possible the detection and trapping of the system calls. This is achieved by placing the `INT3 (0xCC)` byte at the beginning of the system call function. This byte is executed by the central processing unit (CPU) as a debugging interrupt, a breakpoint. This in its turn triggers a VM-exit, which is caught by DRAKVUF and handled by our callback function. DRAKVUF implements multiple EPTs with different permissions for the same page. This allows placing the trap in the system calls, while at the same time when accessed for read, the original functions are accessed, not revealing this way the injected breakpoint.

Table 3.3 shows that there are two generic cases we need to examine. One case is for the `open()`, `rename()`, `link()`, `symlink()`, `unlink()`, and `truncate()` system calls, where we have to find the string pointed by the pointer in the `rdi` register, and in the `rsi` register in the case of `rename()`.

The second case is for the `openat()`, `renameat()`, `renameat2()`, `linkat()`, `symlinkat()`, and `unlinkat()` system calls, where we have to retrieve the string of the file being accessed from different registers, according to table 3.3.

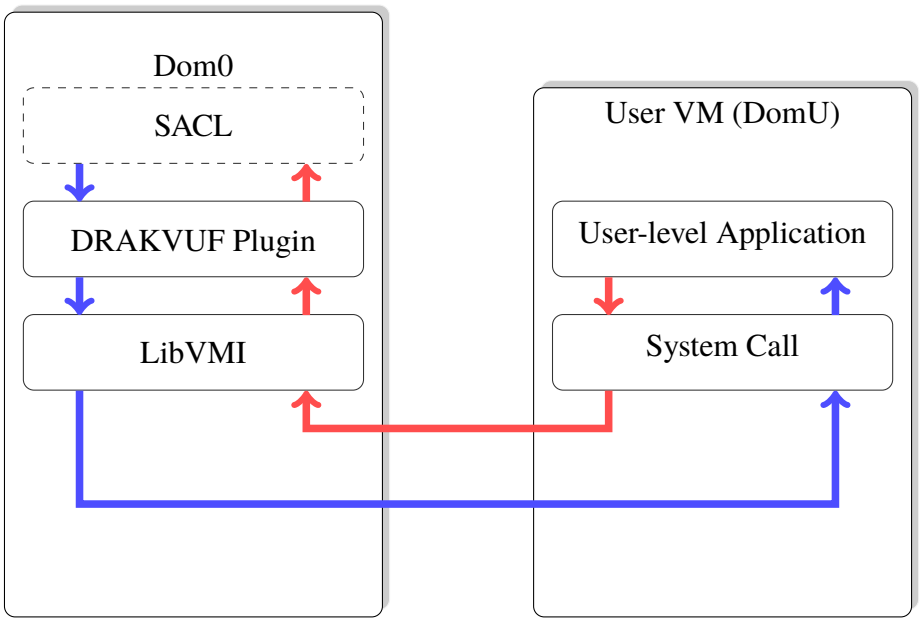

Figure 3.5. Information flow during a trapped system call execution

3.5.5 The task struct

A crucial part in the design of our solution is the Linux kernel `task_struct`. It a complex structure where the kernel stores many information concerning the running processes. Each running process is assigned one such structure by the kernel, so that the kernel can monitor the process and retrieve various information about it. A special macro, `current`, points directly to the current running process. We need to map this structure and find the address offsets of the information we need. We will revisit the `task_struct` in the next sections, as we mention what we need to access.

Although normally this process is not complex, in our case there are some challenges, the semantic gap already mentioned for all out-VM solutions. The first is that we need to find the correct offsets inside the `task_struct` for the entries we want, which depend on the kernel version. Furthermore, we need to make constant conversions between guest machine frame number (GMFN) and machine frame number (MFN), as the memory values we retrieve correspond to the VMs address space, but we need to access the actual physical memory to read the information we require.

3.5.6 Trap Handling

After a hooked system call gets executed, our callback function is called. We firstly retrieve the file that is being accessed and by which process, by getting its process id (figure 3.6).

When a system call is executed the file being accessed is passed as a string pointer. After we retrieve the string, we check whether the file is given in the form of an absolute or relative path. If the path is absolute there is nothing more to do and the algorithm continues. If the path is relative, the retrieval procedure is more complicated. That is because we need to recreate the present working directory (PWD). The Linux kernel does not store this information somewhere. On the contrary, in the `task_struct`, the kernel only stores in another structure only the parent directory. Therefore, we need to loop through the parent folders, so that by prepending each time the parent, we recreate the PWD and the full pathname. To achieve this we created a new function that walks though the codefttask_struct to collect all the required information.

The case for the `openat()`, `renameat()`, `renameat2()`, `linkat()`, `symlinkat()`, and `unlinkat()` system calls is different. Among the arguments, there is one called

```
currpid = vmi_dtb_to_pid(vmi, info->regs->cr3);
switch (info->regs->rax){
case S_OPEN:
case S_RENAME:
case S_UNLINK:
   addr=vmi_translate_uv2p(vmi,info->regs->rdi,currpid);
   filename=vmi_read_str_pa(vmi,addr);
   .
   .
   break;
case S_OPENAT:
case S_UNLINKAT:
case S_RENAMEAT:
case S_RENAMEAT2:
   addr=vmi_translate_uv2p(vmi,info->regs->rsi,currpid);
   filename=vmi_read_str_pa(vmi,addr);
   .
   .
   break; }
```

Figure 3.6. Getting the file being accessed

`dirfd`. This argument is not always used. When it is not, the algorithm is the same as above. When it is used, it is in the form of a directory descriptor. This means that the process has already successfully opened a directory, and passed its descriptor as an argument, which represents the mount directory for the file. In this case we go though the `task_struct` structure of the current running process and retrieve the directory that maps to the directory descriptor, so that we can recreate the pathname of the file.

After we have retrieved the pathname, we then check for the system call that triggered the VM-exit event. For this research we will not handle the `name_to_handle_at()` and `open_by_handle_at()` system calls. At this point of the research we are unaware of any compiled program that uses this specific system call. Support for them can be added in the future, while at the same time does not hinder our proof of concept. The rest of the system calls are handled as follows:

- `link()`, `linkat()`, `symlink()`, `symlinkat()`, `unlink()`, and `unlinkat()`
- `open()` and `openat()`
- `rename()`, `renameat()`, and `renameat2()`

In the first case, where the system calls are used for linking and file deletion the procedure

is straightforward. Once we look in the SACL, if there is an entry, we verify that the user or group accessing the file, has sufficient permissions. If that is true, the callback function returns control to the VM to resume execution. If permissions do not match, the pointer to the filename string is modified to `NULL`, so that the system call, after the VM resumes execution, fails (figure 3.7). By hooking and preventing execution of these system calls we prevent deletion and linking of the protected files, improving this way the availability and confidentiality assurances of the underlying OS.

```
case S_UNLINK:
case S_UNLINKAT:
   switch(info->userid){
      case ROOT: // root user
         if(strcmp(check->pathname,filename)==0){
            check_permissions(check,info,vmi,ROOT);
   .
   .
         break;
      default:  // other users
         if (strcmp(check->pathname,filename)==0){
            if (check->u == info->userid){
               check_permissions(check,info,vmi,USER);}
            else if (check->g == info->groupid){
               check_permissions(check,info,vmi,GROUP);}
         else {
            check_permissions(check,info,vmi,OTHER);}}
   .
   .
         break;}
```

Figure 3.7. unlink() and unlinkat() skeleton code flow

The search for a match in the SACLs is performed in two steps for all cases. Once to go through protected folders and once to go through individually protected files. Also, the `root` user is handled separately from the rest of the users because of the special elevated privileges that account is granted.

In the case of `rename()`, `renameat()` and `renameat2()`, which are used for file moving, we perform the same check as per `unlink()` and `unlinkat()`, with the difference that if we do not find a match for the `oldname` of the system call, we additionally check for a match on the `newname`. The first part ensures that if the user or group does not have read permissions, as enforced by our SACL, he cannot rename or move the file to an unprotected folder or filename, ensuring the *confidentiality* of the information stored in the file. In

the second case we prevent the protected file from being overwritten by another file, if the permissions are not correct. This way we improve *integrity* of the underlying OS, by preventing modification of the protected file.

Finally in the case of `open()` and `openat()`, we only have to check for one filename in our SACLs. If there is an entry, then the permission check algorithm is more complicated. This happens because we have to match the requested by the process access `mode` with the permissions we want to enforce. So, we check for read permission when a `O_RDONLY` access is requested, and for write permissions on a `O_WRONLY`. In the case of a `O_RDWR` request we initially check for both permissions. If that fails, we then check if the process user or group has read permissions. If that is true, we alter the file access mode to read-only and allow execution. If all of them fail we change the `rax` register contents to `NULL` and resume VM execution, which results to a failed system call. This more complex permission check improves confidentiality, by not allowing read access to those who do not have the right, and integrity and availability by denying write access to those who cannot write to the file, as per the SACL enforced policy. Figure 3.8 shows the code for the permission checks done in case of the `open()` and `openat()` system calls.

```
switch(info->regs->rax){
   case S_OPEN:
   case S_OPENAT:
   if ( ((info->regs->rsi & 07) | O_RDONLY) == O_RDONLY){
      if ( !(check->mode & r) ) {
         vmi_set_vcpureg (vmi, 0, RDI, info->vcpu);
         return 1; }
   } else if ( ((info->regs->rsi & 07) | O_WRONLY) == O_WRONLY) {
      if ( !(check->mode & w) ) {
         vmi_set_vcpureg (vmi, 0, RDI, info->vcpu);
         return 1; }
   } else if ( ((info->regs->rsi & 07) | O_RDWR) == O_RDWR) {
      if ( !(check->mode & w) && !(check->mode & r) ) {
         vmi_set_vcpureg (vmi, 0, RDI, info->vcpu);
         return 1;
      } else if ( !(check->mode & w) ){
         vmi_set_vcpureg(vmi, 0, RSI, info->vcpu);
         return 1; }
   }
   break;
}
```

Figure 3.8. open() and openat() permission checks

When one of the trapped system calls gets executed, LibVMI pauses the VM execution. It then passes the VMs state information to DRAKVUF, where our running plugin retrieves it. At this point the guest OS is paused and none of its processes continue running, being unable to bypass and alter the VM's state. Going through some VM memory accesses, the plugin gets the file being accessed, the `userid`, and the `groupid`. With this information, it goes through the SACL to find any matching files or folders that are being protected. If none are found, it returns control to LibVMI, which then resumes the VM's execution. If an entry in the SACL is found, then the plugin checks if the requested file access is prohibited. If it is allowed, execution continues normally. If it is prohibited, on the other hand, then the plugin changes the value of some registers related to the system call so that it will fail.

3.6 Monitoring of program execution

In addition to the system calls of table 3.1, we trapped also the `execve()` and `execveat()` system calls. Following the technique described in 3.5.6, we retrieve the pathname of the program that is requested to be executed. We then check the permissions stored in the SACLs and decide on allowing execution or not.

In addition to checking the SACL permissions, to deny execution immediately if the file is not in the SACL. This creates an efficient application white-list monitoring tool, that can mitigate a broad variety of viruses and malware in general. The reason for this is that most of the malware will save its code on a file for later use. When it tries to execute this file, since it is not part of the original file-system, execution will cancel. We must note that for a malware to create a persistence mechanism on a compromised system, it must write its code on a file. By write protecting the already present executable files, and denying execution of new files, we mitigate a significant volume of old and new malware, including *zero-day* attacks.

3.7 Guest VM Configuration

As mentioned before, there is no significant setup for the guest VM in order for our system to run. The only requirement coming from LibVMI and DRAKVUF is the creation and export of a Rekall profile in the guest VM. Because this profile depends on the kernel version running, it is imperative to recreate the profile in the case of a kernel version update.

To protect the VM from running unprotected in such a case, we have set the options in the Xen guest configuration file, which resides on the hypervisor, to shutdown the VM in case it needs to reboot, as shown in figure 3.9. This does not affect significantly the usability of the guest system, as the Linux OS seldom requires a reboot, even after software updates. Additionally we trap and deny the `kexec_load()` system call, which is used to load a new kernel to be used during the next boot sequence. These two steps protect against custom built kernels, compiled by attackers, since the VM will not run unsupervised, and will not allow the loading of a new kernel. The administrator will need to investigate harder to determine why the VM was rebooted, or powered off in the first place.

```
on_poweroff = "destroy"
on_reboot = "destroy"
on_crash = "destroy"
```

Figure 3.9. Guest VM shutdown configuration line

To support file confidentiality, integrity and availability even from root access, we need to prohibit the root user from executing the `su` command. This command, short for switch user, allows root to switch to any account in the OS. To do that we need to edit a configuration file so that the execution of this command is not allowed. Our test VM uses the pluggable authentication module (PAM). To achieve the required result, we edited the `/etc/pam.d/su` file in the guest OS by adding the line shown in figure 3.10, and then denying the write permission of all users for that file in the SACL so that it cannot be modified.

```
auth required pam_wheel.so deny group=root
```

Figure 3.10. Guest VM configuration to deny *root* from running *su*

Furthermore, since `root` can change other user passwords, we also need to deny that capability. To do that, we do not need any special in-VM configuration. We just need to protect the `/etc/shadow` file from being modified by anyone. Therefore, a sample SACL to enforce these minimum security requirements we have set, is shown in figure 3.4

Concluding, we have seen that we make no alteration at all at the guest VM. Concerning usability, the only things we have restricted are the ability of the `root` user to change the password of any other user, and the capability of the root user to switch to any other user.

This can be troublesome in the event that a user has forgotten his password, but we assess it as an acceptable usability limitation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Evaluation

In this chapter, we present the evaluation of `ferify`. In the first section we validate our design of `ferify`. Next we explain the tests we performed to verify that `ferify` has the results we expect. These tests represent the cases `ferify` can be employed. Finally we perform some tests to evaluate the performance overhead and the degree our solution impacts overall system usability.

4.1 Validation

In 3.1 we discussed why `ferify` is secure. To expand more on this and evaluate that our claim is correct we will present the security challenges we identified and tried to solve during the implementation of `ferify`. We can classify them by the permission level required to perform certain malicious operations on the running operating system (OS).

4.1.1 Ring 3 operations

The ring 3 operations include all the actions an attacker can perform in user-space. Assuming that the attacker has gained `root` access, these actions include modification of system configuration files and programs to serve certain purposes for the attacker.

`ferify` can be employed to overcome this challenge. By adding critical system files in the shadow access control lists (SACLs) along with the correct permissions, we can *deny* write access to any user, *including* `root`. The administrator must decide which files are critical, depending on the security risk that is anticipated. A few basic files we assess that need protection, to ensure some of our assumptions are `/etc/passwd`, `/etc/shadow`, `/etc/pam.d/su`. Tests for these cases are covered in the next section.

4.1.2 Ring 0 operations

File operations are performed by the kernel through the system calls. But the kernel can be modified by the `root` user, who has access to it. Kernel memory can be accessed only

while operating in ring 0 with the following ways:

- System calls,
- Kernel modules,
- Kernel compilation and loading,
- kernel bugs.

Generally, the system call application program interface (API) is well defined and difficult to exploit. System call attacks usually target the system call table, to modify the address of the system call functions with others of the attackers choosing. But to do that the attacker needs to be *able to load kernel modules*. So, this type of attack falls already in the second category, that of the kernel modules.

When an attacker manages to load a kernel module, everything is within reach. The module is running as part of the kernel, in ring 0, and has access to *every* physical and virtual resource available on the system. Kernel modules have access to all kernel functions and structures, and are able to modify them. There are protection mechanisms, like module digital signing, but all these checks are running on the same privilege level. To protect the guest OS from that type of attacks, which can also result in the installation of *root-kits*, we trap and block the `init_mod()` and `finit_mod()` system calls, which make module loading possible, *denying* it this way impossible.

To protect against a modified kernel, we took a simplistic approach, easily implemented with `ferify`. We block the system call `kexec_load()`, which loads a new kernel for later execution. Other than that we can detect the kernel files that are used to boot the system and write protect them from all users. Also, the setup of the virtual machine (VM) will shut it down in case of a reboot, so running automatically on a modified kernel is currently prohibited. We decided to not cover this specific attack vector, since there are solutions like vTPM [36], which implement a more sophisticated way of securing the system's boot procedure and verifying that the code launched by firmware is trusted.

4.2 Testing

To assess that `ferify` works as intended, we assume two basic scenarios. The first is that of an authorized remotely connected user, who should be able to work as before. We want

usability at the user level to remain unchanged. The second scenario is that of a hacked user account. The attacker has managed to exploit a vulnerability of the guest OS and has remote access to the target VM. In this case we assume even the worst case scenario that the attacker has gained `root` privileges. We want to assess whether the files we protect with `ferify` are secure and cannot be accessed except from those we have allowed in the SACLs.

To emulate the actions of an attacker, we performed specific commands on the guest VM in order to observe the behavior of the system and whether it conforms to the expectations. The commands reflect the cases that `ferify` can be applied and correspond to the most basic commands one can issue. All higher level programs and attacks eventually resort to the execution of these system calls. To do that we created an environmental setup to assess all the test cases.

The configuration includes two users, `alice` (authorized user) and `bob` (attacker), both in the *sudoers* group, but `bob` not legitimately, but through malicious actions. The intent is to check the detection of the illegal escalation of `bob`'s account to `root` and the effect it has. A high level overview of our design is illustrated in Figure 4.1.

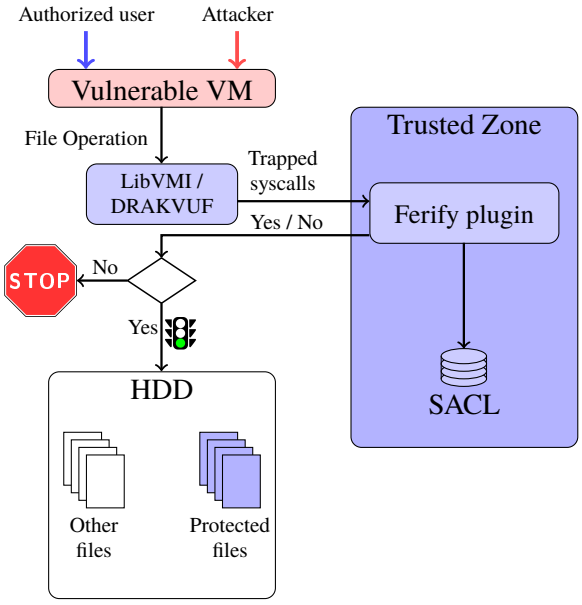

Figure 4.1. Testing and validation concept

Additionally we created a group that includes both users, with the intention to assess the

per group policy enforcement to file access.

Finally we will try, as `root`, to access files that are being protected from this specific user. These files include the `/etc/shadow` file and the `/etc/pam.d/su`, which are included in the SACLs to protect our initial VM configuration.

These cases by themselves cover the basic set of actions that `ferify` was initially designed to monitor. Any combination of them is handled separately from the underlying OS, reducing our problem to these elementary cases, making it easier to monitor and handle.

First we will test if an illegal `root` escalation is possible and then we will test the file operations, which include *create, open, read, write, copy, move, delete, and link*.

4.2.1 Root escalation

One of the first actions of an attacker or malicious program is to try to escalate its privileges to that of the root account. One of those techniques is to add the compromised user to the *sudoers* group. To avoid such an exploitation, every time a trap gets caught by the hypervisor we check whether the user executing a `sudo` command is actually allowed to, according to the initial administrative setup of `ferify` for the specific VM. If the user is a legitimate `sudoer` the flow of our plugin continues normally to check the validity of the file operation against the SACLs. On the other hand, if the user is *not* supposed to have `sudo` privileges, the system calls, trapped by `ferify`, that get invoked get canceled, denying the ability to perform any `sudo` command. Figure 4.2 shows the result of trying to execute a `sudo` command that gets denied, while at the same time we get notified on the hypervisor of the event as shown in fig. 4.3.

```
bob@aHVM-domU:~$ sudo ls
   sudo: unable to open /etc/sudoers: Bad address
   sudo: no valid sudoers sources found, quitting
   sudo: unable to initialize policy plugin
bob@aHVM-domU:~$
```

Figure 4.2. Denying *sudo*

In the same manner the result of trying to change a users password shows in fig. 4.4

Having solved the issue of *malicious root access*, we then need to validate the expected

```
Warning: Process root identity corruption detected in task
         1377! Is 0 and should be 1002.
         Invalidating syscall.
[SYSCALL:   2] CR3:0x1afec000  , RDI: 0x7f2178a931f7 ,sudo
         PID:1377 [0:1002] wants 0 access to file:
         /etc/sudoers (mode:0)
```

Figure 4.3. Denying *sudo* notification on the hypervisor

```
bob@aHVM-domU:~$ passwd alice
   Enter new UNIX password:
   Retype new UNIX password:
   passwd: Authentication token manipulation error
   passwd: password unchanged
bob@aHVM-domU:~$
```

Figure 4.4. Denying password change from *root* account

behavior for file access, whether requested from the `root` user or not.

4.2.2 Authorized user operation

A very important aspect of a security solution is the impact on *usability* for the normal user. The implementation of a secure system that remains fully usable is a challenge. In this test we evaluate whether an authorized user can normally access and work on his files. To test the file operations we create two files `file1` and `file2`. In the SACLs we have entries for three files, to test if the third file can be created. The permissions for the files on the guest OS and the SACLs are explained in table 4.1.

Table 4.1. File permissions for authorized user

File Name	Owner			Group			Others		
	Read	**Write**	**Execute**	**Read**	**Write**	**Execute**	**Read**	**Write**	**Execute**
Guest OS									
`file1`	✓	✓	-	✓	-	-	✓	-	-
`file2`	✓	✓	-	✓	-	-	✓	-	-
`file3`	✓	✓	-	✓	-	-	✓	-	-
Hypervisor User's SACL									
`file1`	✓	✓	-	✓	-	-	✓	-	-
`file2`	✓	✓	-	✓	-	-	✓	-	-
`file3`	✓	✓	-	✓	-	-	✓	-	-

After running our test script to verify the ability to normally access the user's files, the

results (fig.4.5) show that the files are accessible in every mode they are supposed to.

```
$ ./file_tests file1 file2 file3
    Success: Opened file for reading. Able to copy.
    Success: Opened file for writing.
    Success: created file.
    Success: File deleted.
    Success: File moved.
```

Figure 4.5. File operations execution for authorized user.

All basic file operations *are available to an authorized user*. We assess that the usability level for a normal user remains unchanged. By modifying the SACL permissions, we can allow or deny certain operations on user owned or group owned files, according to the desired policy to be enforced.

4.2.3 Attacker operation - `root` access - Immutable objects

One of the biggest problems in information security is the fact that the `root` user or the `administrator` of the system has access to *all* the files and folders on a given system. That is the reason most attacks try to result in escalation of privileges to an administrative account. Although, usually, the administrator is someone trusted and should have access to all OS files for management issues, there are cases that even that person should not get access to certain information. `ferify` can provide that capability, for a normal user to have full access to his files, while at the same time the administrator cannot have full or even limited access to them. By setting the relevant file permissions as per table 4.2, we can *deny* total access to the administrator on any file. This SACL is different than the previous one, allowing for user based policy.

After running our test script to verify the ability to deny access to `root` on the user's files, the results (fig.4.6) show that the files are inaccessible in every mode they are supposed to, according to the permissions we set in the SACL. We assess that denying file access to the `root` user is an important security implementation, as it provides great *confidentiality* assurance, as well as it can nullify many attack vectors on a system, old and new.

A result of `ferify` is that in can be used to create immutable files on a system. By denying write access to everyone, there can be files that cannot be modified by anyone, but are still available and accessible for reading, providing this way great *integrity* assurance.

Table 4.2. File permissions for root

File Name	Owner			Group			Others		
	Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
Guest OS									
file1	✓	✓	-	✓	-	-	✓	-	-
file2	✓	✓	-	✓	-	-	✓	-	-
file3	✓	✓	-	✓	-	-	✓	-	-
Hypervisor root's SACL									
file1	-	-	-	-	-	-	-	-	-
file2	-	-	-	-	-	-	-	-	-
file3	-	-	-	-	-	-	-	-	-

```
$ sudo ./file_tests file1 file2 file3
    Failure: Could not open file for reading. Cannot copy.
    Failure: Could not open file for writing.
    Failure: Could not create file.
    Failure: Could not delete file.
    Failure: Could not move file.
```

Figure 4.6. File operations execution for non-authorized user.

4.2.4 Malware Protection

For the `execve()` and `execveat()` system calls we implemented a slightly different logic, to make `ferify` work as a white-listing security application. The results of our testing seem promising.

We have successfully managed to apply different execution permissions and block application execution on a *per-user* level. The difference from the rest of the system call handling is that *if* the executable *is not* in the SACL we *deny* execution. Since we assume that the system is secure when we launch `ferify`, any file in the file-system should not be malicious. Keeping in mind that for a malware to create a persistence mechanism, it will have to write executable code on the victim system, and run it at some point. Since that file will not be in the SACL, we automatically block that executable from running, having this way mitigated efficiently a large attack vector of malware.

4.2.5 Append-only mode

We could not enforce partial write permissions to a file, due to the OS limitations. A file can be opened as a whole, given the permissions the user requests. What `ferify` is capable

of doing is to enforce a write-only access policy to files. This way, if attackers want to cover their tracks, they are not able to open a file in read-write mode, to select and delete the log entries that reveal their presence. They will have to blindly delete entries, or empty the entire log, action that should be noticed by the system administrator. We believe that this policy can be proven a substantial hindrance to malicious actors that try to cover their tracks in the system.

4.3 Performance overhead

In this section we present the performance overhead we observed on the guest OS when running `ferify`. We performed time execution measurements in three stages.

To have a baseline for comparison, we created a `Python` script that performs some of the trapped system calls and measures the execution time. It performs the execution and calculates the average execution time and the standard deviation. We performed three runs of testing. The first was under normal central processing unit (CPU) scheduling preferences. For the second run we set the `niceness` of the process to `-20`. *Niceness* for Linux is how favorable the scheduling will be for the process, with `-20` being the best value. Finally, for the third run, besides changing the niceness of the process, we additionally set the I/O niceness to the best available value. This changes the I/O scheduling of processes, giving our test script I/O priority over all other processes. Initially we timed the execution of the script over a number of one million iterations to extract an average time that we will use as a reference, *without* running `ferify`. Fig. 4.7 shows the three commands we used to run the test script.

```
$ sudo ./ferify_test.py 1000000
$ sudo nice -n -30 ./ferify_test.py 1000000
$ sudo nice -n -30 ionice -c 1 -n 0 ./ferify_test.py 1000000
```

Figure 4.7. Test script command-line settings.

4.3.1 Hypervisor - VM switch

One of the most intense operations in virtualization is the switch between the hypervisor and the VM, as mentioned in subsection 2.4.1. CPU virtualization extensions improve with every new model release, but this switch is still significant. To measure this overhead in

performance, we ran `ferify` with empty SACLs. This way `ferify` traps the appropriate system calls and performs the switch between the VM and the hypervisor. Because the SACLs are empty, there is insignificant computation performed while on the hypervisor, switching almost immediately back to the VM.

Table 4.3 shows the results and the performance overhead of the VM context-switch. As expected the cost of VM-exit and VM-entry is significant.

Table 4.3. Context-switch performance overhead measurements

	With `ferify`		Without `ferify`		
System call	Avg time (msec)	Std dev	Avg time (msec)	Std dev	Increase
No scheduler priority set					
`open()`	0.139568	0.149873	0.010482	0.011007	1331.50%
`rename()`	0.130636	0.152204	0.003966	0.004409	3293.89%
`unlink()`	0.127103	0.153161	0.003701	0.004258	3434.28%
With best `nice` value					
`open()`	0.139860	0.151600	0.010345	0.010674	1351.95%
`rename()`	0.131188	0.134321	0.004087	0.004329	3209.88%
`unlink()`	0.127132	0.131674	0.003713	0.003966	3426.96%
With best `nice` and `ionice` values					
`open()`	0.137672	0.140117	0.010366	0.011140	1328.11%
`rename()`	0.128105	0.138182	0.004110	0.004440	3116.90%
`unlink()`	0.125161	0.131322	0.003778	0.004190	3312.89%

4.3.2 SACL performance overhead

The information for the files stored in the SACLs is stored in a hash-table. The index of the table is the pathname. This makes storing and retrieving entries from the SACL a complexity of *O(1)*, making the algorithm efficient. Nevertheless, we need to measure `ferify`'s total performance overhead when running on a full file-system. We ran the same script as before, but this time we included a full file-system SACL, to evaluate `ferify`'s efficiency. The SACL contained *417,461* entries.

Table 4.4 shows the result of the second run of our test. We observe that the total performance overhead remains in the same magnitude. We assess that the hash_table implementation and the permission checking algorithm has an insignificant impact on performance, compared to the cost of switching between the hypervisor and the guest VM.

Although the increase in execution time seems extremely large, the average execution time of a single system call seems to remain within usable limits. While using the guest VM to evaluate the use experience, we did not notice any delays.

Table 4.4. `ferify`'s performance overhead measurements

	With `ferify`		Without `ferify`		
System call	Avg time (msec)	Std dev	Avg time (msec)	Std dev	Increase
No scheduler priority set					
`open()`	0.251433	0.258587	0.010500	0.011618	2394.60%
`rename()`	0.297373	0.307451	0.003917	0.004532	7591.85%
`unlink()`	0.240050	0.257848	0.003713	0.004476	6465.12%
With best `nice` value					
`open()`	0.136804	0.146983	0.010478	0.010743	1305.63%
`rename()`	0.127728	0.150706	0.004144	0.004387	3082.23%
`unlink()`	0.125051	0.235790	0.003749	0.004016	3335.58%
With best `nice` and `ionice` values					
`open()`	0.141293	0.143097	0.011300	0.012296	1250.38%
`rename()`	0.131271	0.133869	0.004304	0.005099	3049.97%
`unlink()`	0.128175	0.132942	0.003919	0.004685	3270.60%

CHAPTER 5:
Conclusion and Future Work

- Better access control list (ACL) storing and lookup algorithm **Done**
- Multiprocess applications? Seems like I get sometimes different process base.
- Multithreadded applications.
- Extend plugin to monitor file execution too. Can be done by trapping `sys_execve()` and reading the filename of the program. I think that it is always passed as absolute path, so it should be relatively easy to implement. Syscall number is weird. Needs to check for *both execute and read permissions*.
- Implement on the fly SACL updates.
- Modify code as necessary to work in multi-CPU VMs. Maybe it works?
- Add `readlink()` to prevent reading of other files? Is it necessary?
- Check destroy of `hash-tables`.
- Refinement of what files and folders should be in the SACL. Some, due to group privileges might pose a hindrance to usability. Better leave these out, or even better, just be more precise of what is in. Or we can add the corresponding permission to the OTHER.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] P. Mell and T. Grance, *The NIST definition of cloud computing*. Gaithersburg: NIST, 2011.

[2] T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Ndss*, 2003, vol. 3, pp. 191–206.

[3] B. D. Payne, "Libvmi," Sandia National Laboratories, Tech. Rep., 2011.

[4] 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*, 2014.

[5] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[6] "Xen project software overview," accessed: 2017-05-09. Available: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview

[7] B. D. Payne, D. d. A. Martim, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 385–397.

[8] D. Chisnall, *The definitive guide to the xen hypervisor*. Upper Saddle River, NJ: Pearson Education, 2008, ch. 5.

[9] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: approaches, applications, and evolutions," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 10, 2015.

[10] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.

[11] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 297–312.

[12] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[13] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. Van Doorn, "Building a mac-based security architecture for the xen open-source hypervisor," in *Computer security applications conference, 21st Annual*. IEEE, 2005, pp. 10–pp.

[14] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 74–82, 2008.

[15] J. Xiao, L. Lu, H. Wang, and X. Zhu, "Hyperlink: Virtual machine introspection and memory forensic analysis without kernel source code," in *Autonomic Computing (ICAC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 127–136.

[16] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *ACM SIGARCH Computer Architecture News*, no. 1. ACM, 2013, vol. 41, pp. 265–278.

[17] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ACM SIGARCH Computer Architecture News*, no. 1. ACM, 2008, vol. 36, pp. 2–13.

[18] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[19] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 640–656.

[20] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[21] P. Macko, M. Chiarini, M. Seltzer, and S. Harvard, "Collecting provenance via the xen hypervisor." in *TaPP*, 2011.

[22] M. Crawford and G. Peterson, "Insider threat detection using virtual machine introspection," in *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. IEEE, 2013, pp. 1821–1830.

[23] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski, "Detecting stealth software with strider ghostbuster," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005, pp. 368–377.

[24] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 47–60.

[25] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.

[26] C. Mahapatra and S. Selvakumar, "An online cross view difference and behavior based kernel rootkit detector," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 4, pp. 1–9, 2011.

[27] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 363–374.

[28] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *ACM SIGOPS Operating Systems Review*, no. 6. ACM, 2007, vol. 41, pp. 335–350.

[29] X. Xiong, D. Tian, P. Liu *et al.*, "Practical protection of kernel integrity for commodity os from untrusted extensions." in *NDSS*, 2011, vol. 11.

[30] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Proceedings of the 28th annual computer security applications conference*. ACM, 2012, pp. 429–438.

[31] A. Baliga, L. Iftode, and X. Chen, "Automated containment of rootkits attacks," *Computers & Security*, vol. 27, no. 7, pp. 323–334, 2008.

[32] M. R. Nasab, "Security functions for virtual machines via introspection," Master's thesis, Chalmers University of Technology, 2012.

[33] M. C. Daniel P. Bovet, *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2005, ch. 1.

[34] E. Perla and M. Oldani, *A guide to kernel exploitation: attacking the core*. Burlington, MA: Elsevier, 2010.

[35] "Wrekall memory forensic framework." Available: http://www.rekall-forensic.com/

[36] R. Perez, R. Sailer, L. van Doorn *et al.*, "vtpm: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

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