CLOUDSKULK: DESIGN OF A NESTED VIRTUAL MACHINE BASED ROOTKIT-IN-THE-MIDDLE ATTACK

by

Joseph Anthony Connelly

A thesis
submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Boise State University

December 2017

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Joseph Anthony Connelly

Thesis Title: CloudSkulk: Design of a Nested Virtual Machine Based Rootkit-in-the-Middle Attack

Date of Final Oral Examination: 23 August 2017

The following individuals read and discussed the thesis submitted by student Joseph Anthony Connelly, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Jidong Xiao, Ph.D.	Chair, Supervisory Committee
Catherine Olschanowsky, Ph.D.	Member, Supervisory Committee
Amit Jain, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Jidong Xiao, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

DEDICATION

dedicated to my mother, Terryl Lynn Connelly

ABSTRACT

Virtualized cloud computing services are a crucial facet in the software industry today, with clear evidence of its usage quickly accelerating. Market research forecasts an increase in cloud workloads by more than triple, 3.3-fold, from 2014 to 2019 [33]. Integrating system security is then an intrinsic concern of cloud platform system administrators that with the growth of cloud usage, is becoming increasingly relevant. People working in the cloud demand security more than ever. In this paper, we take an offensive, malicious approach at targeting such cloud environments as we hope both cloud platform system administrators and software developers of these infrastructures can advance their system securities.

A vulnerability could exist in any layer of a computer system. It is commonly believed in the security community that the battle between attackers and defenders is determined by which side can exploit these vulnerabilities and then gain control at the lower layer of a system [22]. Because of this perception, kernel level defense is proposed to defend against user-level malware [25], hypervisor-level defense is proposed to detect kernel-level malware or rootkits [36, 47, 41], hardware-level defense is proposed to defend or protect hypervisors [4, 51, 45].

Once attackers find a way to exploit a particular vulnerability and obtain a certain level of control over the victim system, retaining that control and avoiding detection becomes their top priority. To achieve this goal, various rootkits have been developed. However, existing rootkits have a common weakness: they are still detectable as long as defenders can gain control at a lower-level, such as the operating system level, the hypervisor level, or the hardware level. In this paper, we present a new type of rootkit called CloudSkulk, which is a nested virtual machine (VM) based rootkit. While nested virtualization has attracted sufficient attention from the security and cloud community, to the best of our knowledge, we are the first to reveal and demonstrate nested virtualization can be used by attackers for developing malicious rootkits. By impersonating the original hypervisor to communicate with the original guest operating system (OS) and impersonating the original guest OS to communicate with the hypervisor, CloudSkulk is hard to detect, regardless of whether defenders are at the lower-level (e.g., in the original hypervisor) or at the higher-level (e.g., in the original guest OS).

We perform a variety of performance experiments to evaluate how stealthy the proposed rootkit is at remaining unnoticed as introducing one more layer of virtualization inevitably incurs extra overhead. Our performance characterization data shows that an installation of our novel rootkit on a targeted nested virtualization environment is likely to remain undetected unless the guest user performs IO intensive-type workloads.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

MITM – Man-in-the-Middle

RITM – Rootkit-in-the-Middle

OS – Operating System

VM – Virtual Machine

VMM – Virtual Machine Monitor

SVM – Secure Virtual Machine

AMD – Advanced Micro Devices

VT – Virtualization Technology

EFER – Extended Feature Enable Register

MSR – Machine Specific Register

VMBR – Virtual Machine-based Rootkit

VMCS – Virtual Machine Control Structure

API – Application Program Interface

KVM – Kernel-based Virtual Machine

QEMU – Quick Emulator

IT – Information Technology

AWS – Amazon Web Services

BP – Blue Pill

CVE – Common Vulnerability and Exposure

IDS – Intrusion Detection System

VMI – Virtual Machine Introspection

NIST – National Institute of Standards and Technology

SaaS – Cloud Software as a Service

PaaS – Cloud Platform as a Service

IaaS – Cloud Infrastructure as a Service

L0 – Level 0

L1 – Level 1

L2 – Level 2

BIOS – Basic Input/Output System

CFQ – Complete Fair Queuing

VGA – Video Graphics Array

DIMM – Dual In-line Memory Module

HW – Hardware

Chapter 1

INTRODUCTION

1.1 Problem Context

The benefits of cloud computing are well established for businesses, organizations, and end-users. Despite the overwhelming advantages of cloud computing, one of the most significant problems with this fast growing computing model is security. While both researchers and cloud defenders continue to develop innovative advancements that improve cloud security, these platforms remain vulnerable to inevitable, unidentified weaknesses. It is the goal of our research to then help identify one such vulnerability before malicious attackers can take advantage of these systems.

Cloud computing conveniently hides many complexities from its users, providing all the benefits of a data center while circumventing nearly all its associated costs: office space, power, cooling, networks, bandwidth, servers, storage, sophisticated software, and highly skilled system administers who configure, install, update, and run these systems. The computing resources provided by cloud platforms are available at any time simply through a network connection. These resources are highly scalable, highly configurable, and typically are costed as a metered pay-as-you-go or monthly payment service. The cloud computing model is transforming the software industry.

In spite of cloud computing's highly influential role over the software industry, there are major barriers that prevent the model from a broader adoption. The software infrastructure and physical hardware responsible for hosting cloud services are shared by all of its users; making user data isolation, loss, and integrity against malicious entities difficult. According to a market survey in 2016, “general security concerns” with cloud services were the number one barrier for cloud adoption by businesses with 53% of these organizations sharing this concern [9]. Although difficult to quantify, cloud security incidents are reported to government authorities and the public every year, validating these concerns. Even the leading cloud computing vendors in 2016 — Amazon holding 45% market share, Windows Azure with 39%, and Google with 18% [9] — are subject to the fact that these platforms are not perfectly secure.

Many applications that manage compellingly private and important user data exist today on the Cloud despite the risk caused by these imperfect systems. For example, Google’s cloud application Google Compute Engine provides Cloud Infrastructure as a Service (IaaS) services for businesses across the world; customers like Spotify, Coca-Cola, BestBuy, Motorola, HTC, and hundreds more. GCE serves many industries ranging from Healthcare, to Technology, to our most notable: Financial Services. Take for instance the online payment service mobile application for smartphones, mCash. The application hosted on GCE, released in February 2014, available today in the AppStore and Google Play, publically reported in September 2016 that it has 430,000 unique users [14]. This virtual bank allows users to send and receive money, pay bills, and view transactions. Many real world cloud applications like mCash, who demand security for their users, are potential targets for a new type of malicious security attack which we call CloudSkulk.

1.2 Thesis Statement

In this thesis we present a new type of software security attack methodology related to the well-known Man-in-the-Middle (MITM) type attacks. Our methodology uses a nested QEMU/KVM Virtual Machine (VM) based strategy we call CloudSkulk. Our claim is that a VM, controlled by a malicious source, can represent a rootkit that can unnoticeably eavesdrop on data communication between QEMU/KVM guest-host pairs on a GNU/Linux cloud platform providing IaaS cloud services. Through our attack, we can gain 100% visibility of the data and live interactions of a targeted guest running in one of these virtualized environments. One of the defining features of our new type of attack is that it can allow an attacker to remain undetected for long periods of time. Our intention is not to increase cloud security concerns, but rather to raise awareness and identification for cloud service vendors and developers of their system's vulnerabilities in the hopes that they strengthen their security.

1.3 Contributions

The major contributions of our work are summarized as following:

1. We present the design and implementation of a new type of rootkit, nested virtualization based rootkits. To the best of our knowledge, we are the first to demonstrate how nested virtualization can be used by attackers for developing rootkits.

2. We characterize the incurred system performance degradation caused by our unique type of rootkit in an effort to quantitatively express our rootkits's ability to remain unnoticed by target guest and host.

3. We provide a sample demonstration of our rootkit installation on a GNU/Linux based machine supporting the QEMU/KVM virtualization infrastructure.

1.4 Thesis Organization

The remainder of this paper is structured as follows. We describe the necessary background information pertaining to virtualization and the cloud in Chapter 2. We then discuss the related work that provided the foundation for our idea behind our attack in Chapter 3. In great detail, we define the design and implementation of an installation of our rootkit in Chapter 4. We present our characterization results in Chapter 5 as well as a discussion about possibilities for defending against our type of attack. And finally, we summarize the impact of our work as it applies to cloud system security and the software security research community in Chapter 6.

Chapter 2

BACKGROUND

2.1 Cloud Environments

Modern data centers are facilities containing tens of thousands of computers with significant network bandwidth capabilities. They maintain large sets of Information Technology (IT) equipment that is used for providing both hardware and software services for third party users, connected through a telecommunication network [42]. Modern cloud computing differs from traditional data centers in that the services provided by the cloud are given to the user through a virtualized environment called a Virtual Machine (VM). We will use a standard terminology throughout this paper to refer to a “host” as a physical cloud computing machine responsible for enabling virtualization for its third party “guest” users.

Virtualization is the foundation of cloud computing. Virtualization refers to the creation of a virtual version of some physical resources, such as an OS, a server, or a device. Programs running from within a guest VM typically execute on a host with the same level of control as other user applications in what is known as the “lowest of privileges.”

Executing programs are permitted access to a machine’s resources based on an execution privilege category system. In order to manage and control groups within this categorical system, modern x86 CPU instruction sets call these different groups rings. The execution privileges range from the highest amount of control in the most privileged ring 0, to the least control in the lowest privilege level rings 3. Figure 2.1 illustrates this privilege concept.

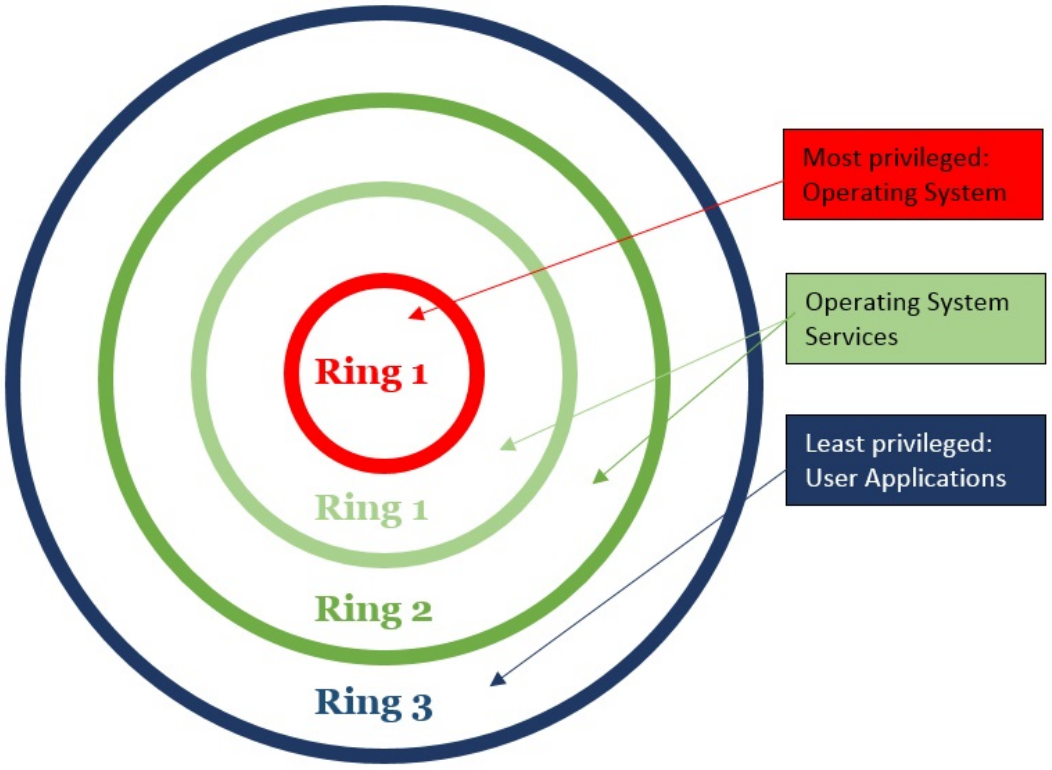

Figure 2.1: CPU Instruction Set Execution Privileges

Virtualized guest programs however may need to access host hardware or privileged host resources requiring, ring 0, authority. To maintain the integrity of system security while allowing the use of these resources, modern computers avoid granting ring 0 level of access to user level (ring 3) VMs through a software work-around accomplished by a Hypervisor, also referred to as a Virtual Machine Monitor (VMM).

Hypervisors provide virtualized guest processes host resources through the various types of virtualization they grant: partial, para-, and full-virtualization to name a few. A hypervisor is essentially low-level, high privileged software that controls the execution of one or more guest OSs on a single machine. Fundamentally, hypervisors function by intercepting and emulating sensitive operations originating from within the guest environment (such as changing page tables, which could give a guest access to host memory it is not allowed to access) [2].

When a privileged command, OS instruction, or non-virtualizable instruction is requested by a para-virtualized guest process, it is delivered to the hypervisor through a HyperCall, similar to Linux system calls. The hypervisor receives these HyperCalls, accesses the hardware, then returns the result. This type of virtualization however requires that the guest OS be modified to properly use the HyperCall API.

A guest created through full-virtualization is provided a complete simulation of the underlying hardware, allowing the execution of an unmodified guest OS to run with privileged, ring 1, control on the host. Full-virtualization can be thought of as using an entire computer, with complete control, encapsulated and running inside a VM. Full-virtualization is slower than other types of virtualization [3] like para-, because machine language code from the running guest OS must, at run-time, be converted to the machine language of the host inside the host hypervisor. This can be accomplished through a process called binary translation if the host CPU supports Virtualization Technology (VT).

Binary translation translates a source binary program to a target binary before execution, where the requested instruction set or binary refers to the source, and the underlying host processor instruction set refers to the target. While sensitive and non-virtualizable instructions of a full-virtualized guest OS (running with ring 1 privileges) are translated using binary translation in the host hypervisor, guest VM user level processes (running with ring 3 privileges), are directly executed on the host CPU for high performance. Modern Intel and AMD CPUs provide this support through assembly language set extensions known as Intel VT-x for the x86-based Intel CPUs, and AMD-v for the AMD type of CPUs. These extended commands allow for a CPU to provide hardware based virtualization. Figure 2.2 illustrates these main functional differences between types of virtualization.

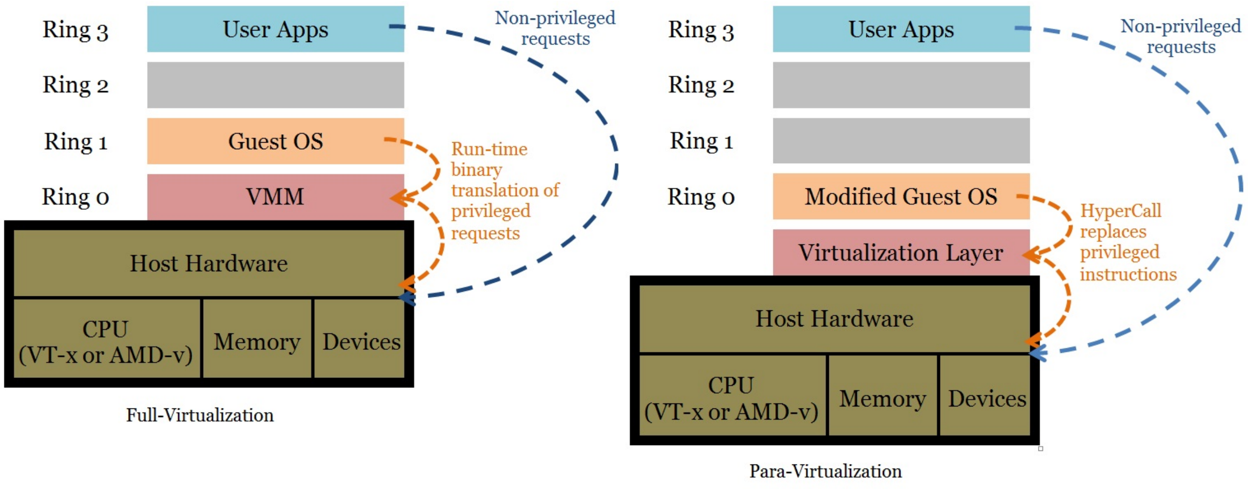

Figure 2.2: Para- vs. Full-Virtualization [18]

2.1.1 Cloud Services

As defined by the National Institute of Standards and Technology (NIST), cloud computing is composed of three fundamental service models: Cloud Software as a Service (SaaS), Cloud Infrastructure as a Service (IaaS), and Cloud Platform as a Service (PaaS) [32].

SaaS cloud platforms provide guests with software. The focus of these cloud platforms is on applications, rather than hardware. The provided software often requires strict and expensive software licenses which are conveniently bypassed by guests who rent the software from the cloud as a SaaS service. As an example, the Microsoft Azure cloud platform provides a popular office suite application called Microsoft Office 365 to users as a SaaS service.

The process of connecting to, and using the services of the Azure cloud is nearly a ubiquitous form of guest-host interaction for cloud platforms. Therefore, examining this process illustrates how cloud services are typically provided to guests in nearly any cloud environment today. An Azure guest first selects a price per hour plan on the Microsoft website based on a set of desired configurations and requested services. The guest then connects to Microsoft's cloud through a local Azure portal application on their local machine in which they can then use Microsoft Office 365, Dynamics 365, or any other requested application from within a VM environment. The Azure guest VM environment is likely to be created and initially maintained within a single process physically running on a cloud machine at Microsoft. The performance of the running application in the cloud is then predominantly dependent upon the hardware and software of the cloud machine. It is important to emphasize that this is one such prevailing benefit of the cloud; that a rented application running on the cloud is likely to run faster than if ran locally. If an expensive CAD application requiring significant CPU processing power can experience performance increases while running in the cloud, CAD users might be persuaded by this cloud attribute.

PaaS based cloud platforms provide a set of software and hardware components that enable guests to build, manage, and run applications on the cloud through a VM. The focus of these cloud platforms are frameworks offered as a service, rather than single applications or individual hardware components like memory rented as a service. A big difference between PaaS and SaaS services is that PaaS is intended more towards effectively and efficiently hosting customized applications for its guest user who are typically themselves businesses or organizations. These businesses or organizations using PaaS services on the cloud then typically deploy their application for other end users of their product. In regards to SaaS cloud based platforms, their guest users are generally just single end users.

The Google App Engine is a cloud platform that serves as a good example of a cloud providing PaaS services. The Google App Engine provides its guests scalable web application services and mobile backends with a framework that includes tools such as: NoSQL datastores, memcache, and a user authentication API. A mobile gaming application company, Pocket Gems, is one such guest using PaaS services to create and deploy its games for other end users. The Google Cloud hosts these services, runs Pocket Gem's games, and provides the infrastructure for a dynamic number of end users. In fact, one of the significant benefits of PaaS services is that hosted applications can be automatically scaled in response to large demands of computation and network traffic. Take another company Niantic, running on Google's cloud, released a mobile application in 2016 called PokemonGO. Google reports that, "Within 15 minutes of launching in Australia and New Zealand, player traffic surged well past Niantic's expectations." [28] A clear illustration of the unexpected levels of PokemonGo player traffic and the dynamic power of the cloud computing model can be seen in Figure 2.3.

While this scalability factor and dynamic quality of cloud computing, seen in Figure 2.3, is true for SaaS services, large scale applications intended for an initially unknown number of end users benefit most from PaaS services. These benefits are one of the central motivations for businesses and organizations to move from local, expense, and internally owned and operated server infrastructures to PaaS service

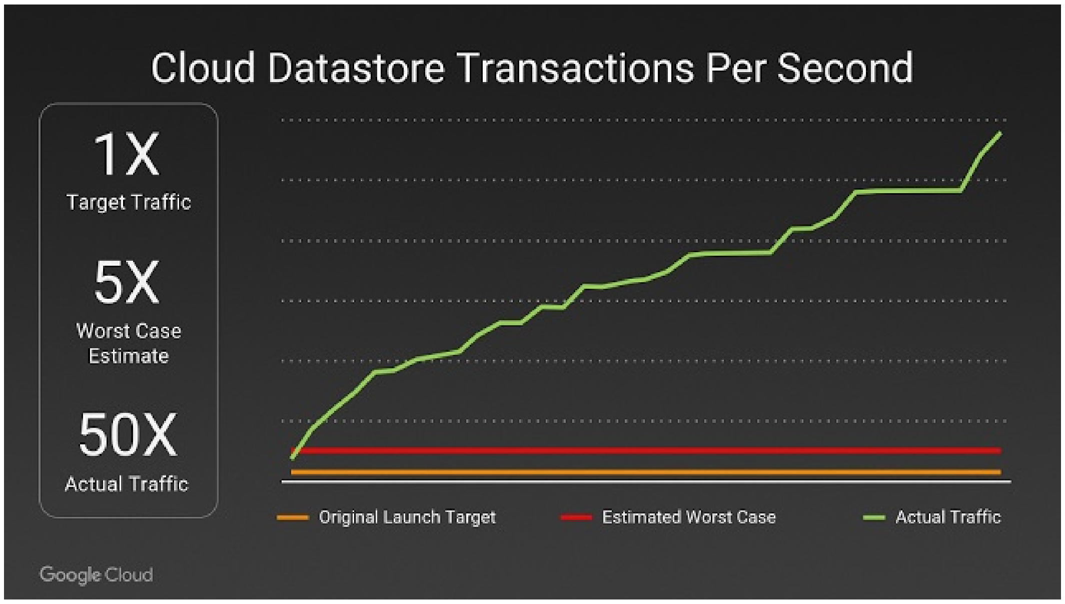

Figure 2.3: Google Cloud Scalability Response to PokemonGO

based cloud platforms.

IaaS cloud platforms provide their guests a special kind of virtualized environment known as a full-virtualization [1]. IaaS based cloud service providers give users an entire computer — user space applications, OS, and hardware — virtualized and accessed through the cloud. Like guest users of PaaS services, guests of IaaS services are typically, although not strictly limited to, themselves being businesses or organizations. As an example, Amazon Web Services (AWS) provides IaaS full-virtualization services for a notable guest that is a world leader in providing online streaming media and video-on-demand content, Netflix. Netflix is an entertainment business that uses IaaS services on AWS to host online content to a reported 98.75 million end users as of April of 2017 [46]. Figure 2.4 illustrates a guest user vs. cloud platform perspective [18] of the services each control and manage with respect to the types of cloud computing

[1] A detailed explanation of full-virtualization is provided in Section 2.1

models described in this section.

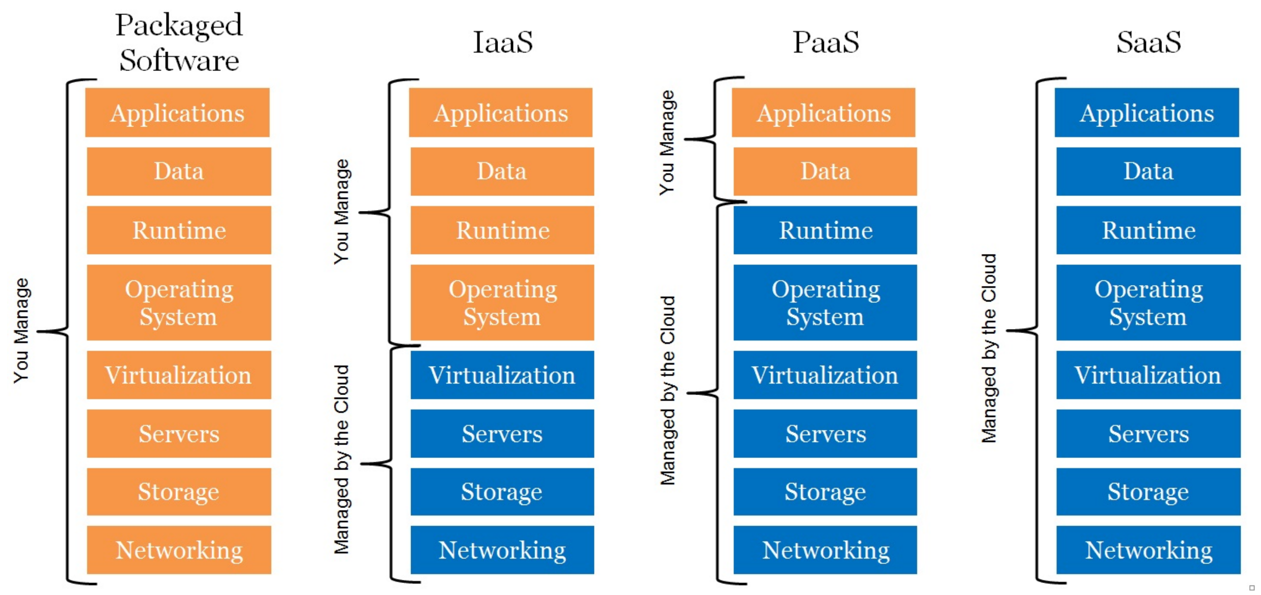

Figure 2.4: User vs. Cloud Resource Responsibility in the Cloud

2.1.2 Nested Virtualization

Although virtualization began in the 1960s, it wasn't until 2010 that researchers from IBM presented the concept of nested virtualization, for the first time, on x86 architectures in their Turtles project [6]. They implemented nested virtualization in the Linux KVM hypervisor [2]. Later on, nested virtualization had been adopted and implemented in Xen (starting from Xen 4.4).

The concept of nested virtualization is straightforward: running a hypervisor inside a VM. Traditional virtualization involves multiple OSs are running on top of the same hypervisor simultaneously. Nested virtualization differs from this model by allowing multiple hypervisors to run on top of the same hypervisor simultaneously. In

[2]A detailed explanation of the Linux KVM hypervisor is provided in Section 2.2

the Turtles project [6], the group of researchers introduced the virtualization concept of Level0 (L0), Level1 (L1), and Level2 (L2) that we will be using in this paper. Level0 represents the hypervisor that runs on top of the real hardware, Level1 represents the hypervisor that runs on top of Level0, as a guest, and Level2 represents the hypervisor that runs on top of the Level1 as a nested guest. Figure 2.5 illustrates these levels of hypervisor virtualization.

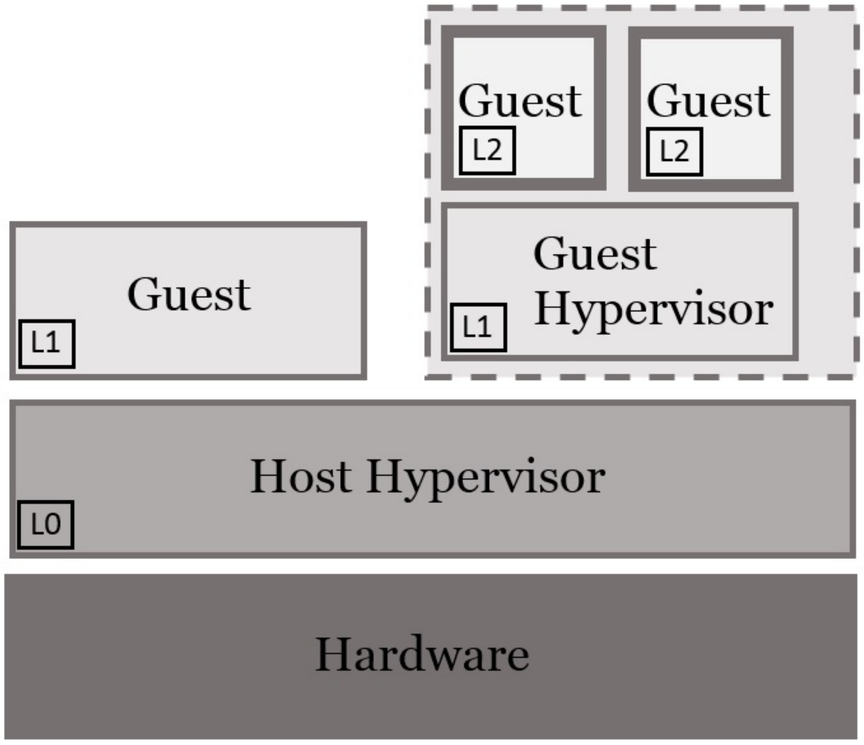

Figure 2.5: Hypervisor Levels of Virtualization

The Turtles project was developed without sufficient CPU architecture support for nested virtualization. Since then, CPU vendors have developed various features in their products to support nested virtualization in the hope of improving nested virtualization performance. For example, Intel introduced the VM Control Structure

Shadowing (VMCS Shadowing) feature in its Haswell processors. This feature extends the capabilities of Intel-VT, with the particular emphasis of eliminating VM-Exits caused by VMREAD and VMWRITE instructions executed by the Level1 hypervisor. Because of these VMCS Shadowing improvements, the frequency of context switches between Level1 and Level0 hypervisors can be considerably reduced, increasing the performance of nested virtualization.

One of the two main design restrictions imposed on a CloudSkulk rootkit [3] is that a guest VM environment is provided with nested virtualization capabilities. In this paper, we target IaaS cloud based platforms as it is these types of cloud services that conventionally offer guests full-virtualization, and therefore by virtue nested virtualization services. It is this nested virtualization feature, as we will see in the following sections, that we can exploit to use for a malicious attack against IaaS based cloud platforms using our unique type of rootkit.

2.2 KVM/QEMU Virtualization

We target the widely adopted Linux hypervisor Kernel-based Virtual Machine (KVM) and its user-space application Quick EMUlator (QEMU). Many leading businesses in the IaaS cloud computing software industry currently employ KVM/QEMU to host their cloud platform services. Google Compute Engine (GCE), Amazon Web Services (AWS), and IBM SmartCloud Enterprise are a few such popular cloud platforms worth identifying.

KVM is a mainstream Linux hypervisor that establishes the software foundational support to enable user-space applications to utilize a computers physical hardware

[3]The entire set of restrictions that were discovered and design decisions that were made for a successful installation of a CloudSkulk rootkit is detailed in Chapter 4.

without allowing user-space applications the ability to interface directly with physical devices. KVM is a type of hypervisor that is called a user-space device emulation architecture [20], meaning it provides only the support for device emulation not the actual creation and execution of such devices. KVM is installed in the Linux operating system by default as a kernel module in versions 2.6.20 and greater.

QEMU is a user-space software application that is responsible for creating instances of virtualized devices as well as the environment in which a guest OS runs on top of. QEMU creates a Linux process for each VM instance, with each process encapsulating the entire VM environment. Virtualized environments using KVM/QEMU can experience near-native performance for typical workloads as demonstrated by Virtual Open Systems [43]. For brevity, we may refer to the KVM/QEMU paradigm as only QEMU for the remainder of this paper. The list below details some the key attributes of these two software tools that apply to our rootkit.

QEMU:

- Modifiable guest OS allowing for nested hypervisors
- Modifiable guest OS (full-virtualization) provides rich set of resources for an attacker to decrease security signatures and anomalies that may lead to detection
- Guest OS is encapsulated from host machine in VM environment
- Nested Guest OS(s) are encapsulated from host machine in VM environment
- Guest VM is created and maintained within a single host Linux process
- Instances of virtualized hardware created for guest OS

- Para-virtualized hardware can be created to increase performance of guest OS
- Supports user-space live migration utility [4]
- Industry standard Virtualization software for IaaS based cloud platforms

KVM:

- Contains implementation of host hardware devices for full-virtualization
- Exposes device emulation to user-space programs through KVM API
- Industry standard hypervisor for IaaS based cloud platforms

2.2.1 Live Migration

In their pioneering work, Clark et al. [8] proposed and implemented VM live migration. They defined VM live migration as a procedure of migrating an entire OS and all of its applications as one unit from one host machine to another host machine. The benefits of VM live migration mainly lie in two aspects: (1) allowing a clean separation between hardware and software and (2) facilitating fault tolerance, workload balance, and low-level system maintenance. They implemented a pre-copy live migration solution in a Xen based virtualization environment, and they argued that the two most critical metrics are: total migration time and service downtime. The former, also known as end-to-end time, refers to the total time taken between the initiation and the completion of a migration, while the latter indicates the duration

[4]Live migration will be defined in length in the next section, Section 2.2.1

when the VM is suspended and the service is not available. Figure 2.6 provides a graphical representation of pre-copy VM live migration.

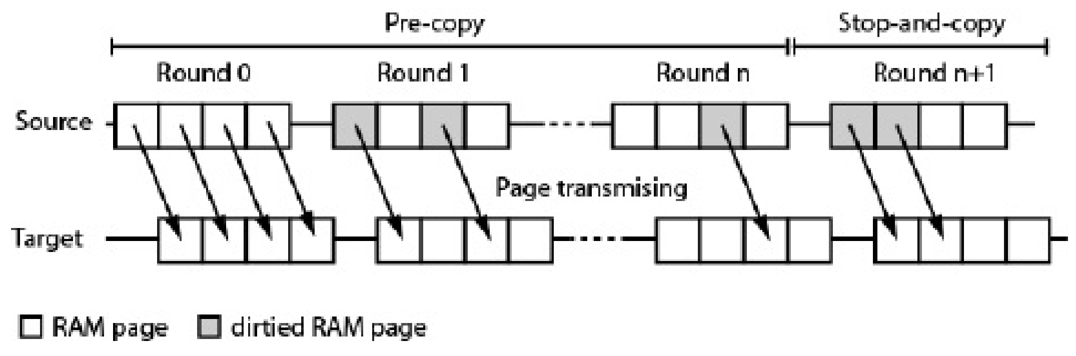

Figure 2.6: Pre-copy Live Migration Algorithm [40]

Since the introduction of VM live migration in 2005, migration has attracted considerable interests from the virtualization and cloud computing communities, and has become a critical feature in mainstream hypervisors, including Xen, Qemu/KVM, VMware, and Hyper-V. Typically, a pre-copy based VM live migration involves four steps, as detailed below.

STEP 1: PREPARATION. In this step, the two host machines, one serves as the source and the other serves as the destination, should both enter into a ready-for migration mode. In particular, the destination side should be listening on some specified port, waiting for a migration request issued from the source side. Once a migration request issued by the source is received by the destination, the two sides will start establishing a TCP connection.

STEP 2: ITERATIVE COPY. Once the TCP connection is established, the memory (and/or disk) of the VM will be copied in an iterative fashion. In the first iteration, all the pages will be transferred from the source side to the destination side. After that, all the pages that get dirtied during the first iteration, will be then copied to the destination, and such a procedure will be repeated iteratively. Such an iterative copying may affect the system performance as it consumes CPU cycles, memory, and network bandwidth.

STEP 3: STOP-AND-COPY If the guest's interactions cause frequent and intensive memory and disk image changes, the VM first needs to be stopped on the source side, then the remaining dirty sections of memory and disk image can be fully copied. This step is where the service downtime comes from. If this remaining part is large enough, the service downtime might be significant.

STEP 4: ACTIVATION Once the copy operation is done, the VM on the destination side will be activated. On the source side, the VM will be paused, and the source host machine can discard the VM.

The above steps depict the basic procedure of the pre-copy live migration. In the latest versions of hypervisors, a post-copy live migration approach has also been implemented. Generally, post-copy incurs smaller downtime, but pre-copy is more reliable. In this paper, we use pre-copy, but cloud vendors could use either pre-copy or post-copy. The rootkit technique we present in this paper applies to both migration approaches. Also, the migration technique detailed above is typically between two physical machines, but this is not strictly the case. As an example, live migration can be used by a host to make dynamic adjustments to guest virtualized devices during run-time. A guest VM with 4G of virtual DIMM memory can be live migrated to a new guest VM with 8G of appended virtual DIMM memory. In this work, including our design, implementation, and evaluation, we only need one physical machine to launch a nested VM based rootkit. In other words, two physical machines are not required in the development of CloudSkulk.

2.2.2 Software Security

On a typical Linux cloud computing machine hosting QEMU virtualized services, KVM runs with privileged super-user permissions, and QEMU runs with less privileged user-space permissions. Guests inscribed within QEMU processes have no permissions on, or access in the host machine unless explicitly initialized as such when the QEMU process is created on the host. However, if a guest process identifies a bug and can exploit it, such as the virtunoid: breaking out of KVM exploit in 2011 [13], the guest can gain partial or even full control over the host machine. Although difficult to quantify, cloud security incidents are reported to government authorities or the public every year. In fact, publicly reported QEMU bugs are reported in Common Vulnerability and Exposure (CVE) bug reports on CVEDetails.com [34]. VM and VMM software bugs vary in degree of severity and are expressed throughout the software security research community [13, 11, 50, 23, 34, 49]. These independent groups were each able to create what are known as Virtual Machine Escapes, allowing full control of their target host OS.

Aside from this limited type of access control of a host machine, many books have been written that discuss how to exploit security bugs and penetrate computer systems and software [30, 24, 29]. It is also well-known for a given network that combinations of security conditions and low-level vulnerabilities can, and do, lead to host control. The most common form of security attack that threatens possible host control through network vulnerably is buffer overflows [10]. This can also be illustrated by the array of ongoing research that seeks to detect and prevent these types of network attacks [7, 37, 44, 35, 27]. Figure 2.7 and Figure 2.8 depict visual representations of two typical types of attacks that gain host control, as described above: Virtual Machine Escape attacks, and host network vulnerability type attacks. In both figures, dotted black arrows represent normal communication paths - dotted red lines represent an extension from this normal use, depicting a threat mode.

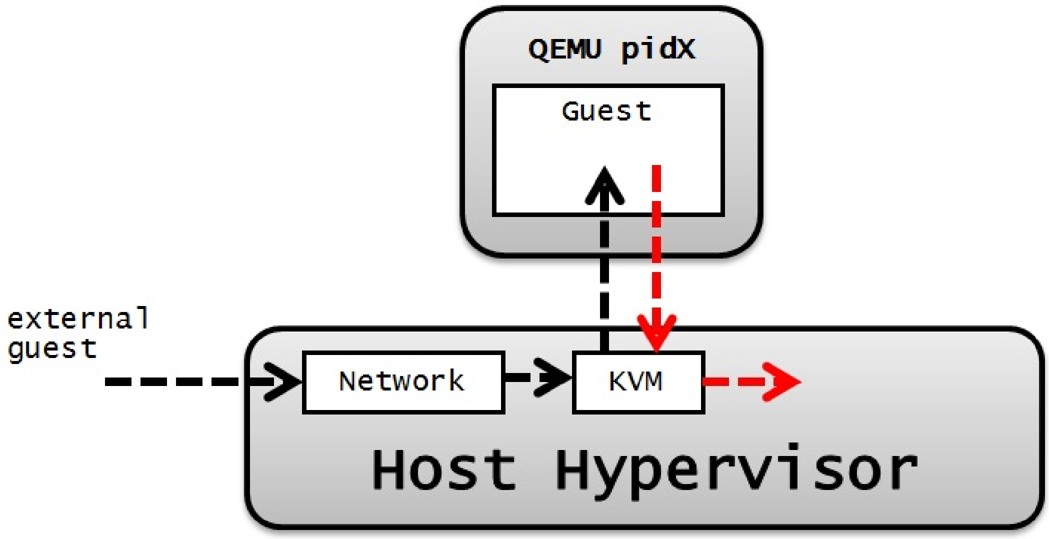

Figure 2.7: Host Intrusion Thread Mode 1 - Virtual Machine Escape

Our malicious attack is possible within the context of either of these types of thread mode events. We will present our design and implementation (Chapter 4) in the context of threat mode 1, making the strong assumption that we are a QEMU guest process that has gained Virtual Machine Escape. It is important to note though

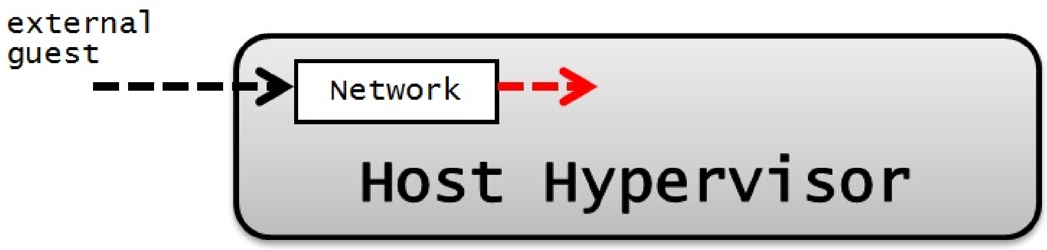

Figure 2.8: Host Intrusion Thread Mode 2 - Network Vulnerability

that any valid attack that gains host control will allow our attack to be implemented. A guest who has gained Virtual Machine Escape can not only invoke commands from within the host, but also attempt to protect own internal VM data by attacking the host hypervisor and/or host Intrusion Detection System (IDS) tools [15].

The characteristics of a compromised host machine provide a powerful software infrastructure for implementing a well-known paradigm of isolated, hard to detect, and potentially malicious programs known as Rootkits. Although not all rootkits are malicious, they can be defined as a small set of programs that allow for a consistent, prolonged, and undetectable privileged access on a computer. The common strategy for maintaining privileged access is through stealth. Rootkits aim to hide code and/or data on a computer system in attempt to remain undetected. It is this crucial characteristic of stealth that we can then relate a virtual machine as conceptually equivalent to a rootkit. This is a fundamental association that we use — that a VM can act as a rootkit — and a key conceptual attribute of our unique type of rootkit.

The well-known and studied Man-in-the-Middle (MITM) type of software security attacks can be classified into two distinct types: passive, and active. Passive MITM attacks are defined by a malicious party, the "Man", whom eavesdrops on messages sent between one or more pairs of users. Active MITM attacks are similarly defined by a malicious party whom - in addition to eavesdropping - modifies messages sent between one or more user pairs [21]. Figure 2.9 is a visual representation of an MITM type of malicious attack, with dotted black arrows representing normal communication paths and dotted red lines representing an un-permitted path of communication controlled either passively or actively by a third party "Man".

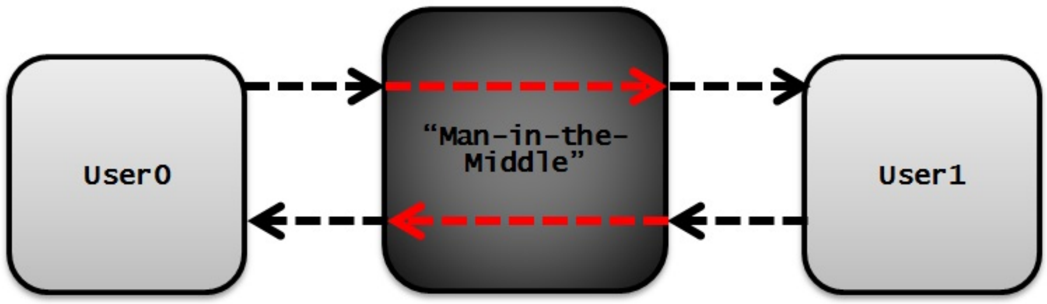

Figure 2.9: MITM Software Security Attack Model

Using the association that a VM can act as a rootkit, we propose a novel association: that a VM can act as the "Man" in a MITM type of attack between a host and nested guest user. When relating MITM concepts to a VM it is useful to illustrate that an inherent function of a virtual machine is to pass messages between a guest-host pair. It then becomes clear that a VM representing a rootkit — the "Man" — can easily, and consistently eavesdrop on communication between a host machine and a nested guest user of a VM. It is also valuable to illustrate that a nested VM based rootkit naturally achieves another main objective for a rootkit, being that it can remain undetected for extended periods of time by simply emulating the virtual environment for the nested guest. In this type of rootkit, a source side VM can also seek to hide any behavior associated with its eavesdropping activities by modifying internal VM software and/or data. Figure 2.10 is an abstraction that depicts a typical virtualization software/hardware set-up for network and virtualization communication flow between a guest-host QEMU/KVM pair. In the figure dotted black arrows represent a guest user's data communication flow of network packets.

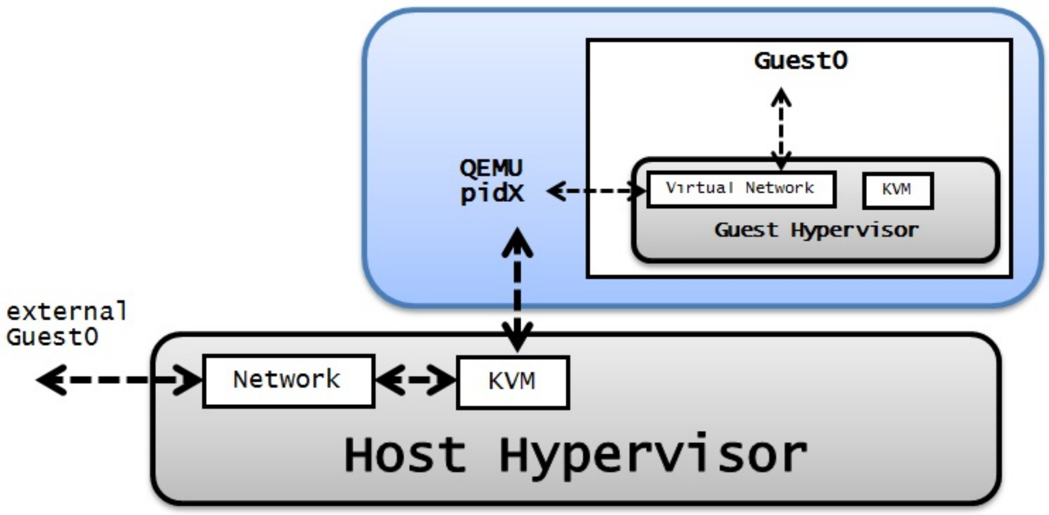

Figure 2.10: KVM/QEMU Virtualization Architecture

Appending our two thread model associations: (1.) that a VM can act as a rootkit, and (2.) a VM can act as a "Man" in a nested VM based MITM attack, we can then create a visual representation of this conceptual model. Figure 2.11 contains a single host side VM (QEMU pidX) which contains in itself a nested guest VM (QEMU pidY), therefore pidX can represent a rootkit. The rootkit is controlled by Guest0, and the nested guest user is controlled by Guest1, both connected to the host over an external network connection. If Guest0 were a malicious user, Guest0 could seek to passively or actively target two victims in a MITM type attack: the host machine, and Guest1. Again, given this nested virtualization architecture setup, Guest0 can represent a malicious "Man" in the middle between the host and Guest1. The full design and implementation details for this type of attack will be provided in Chapter 4, as this is simply an introduction into the conceptual foundation of CloudSkulk.

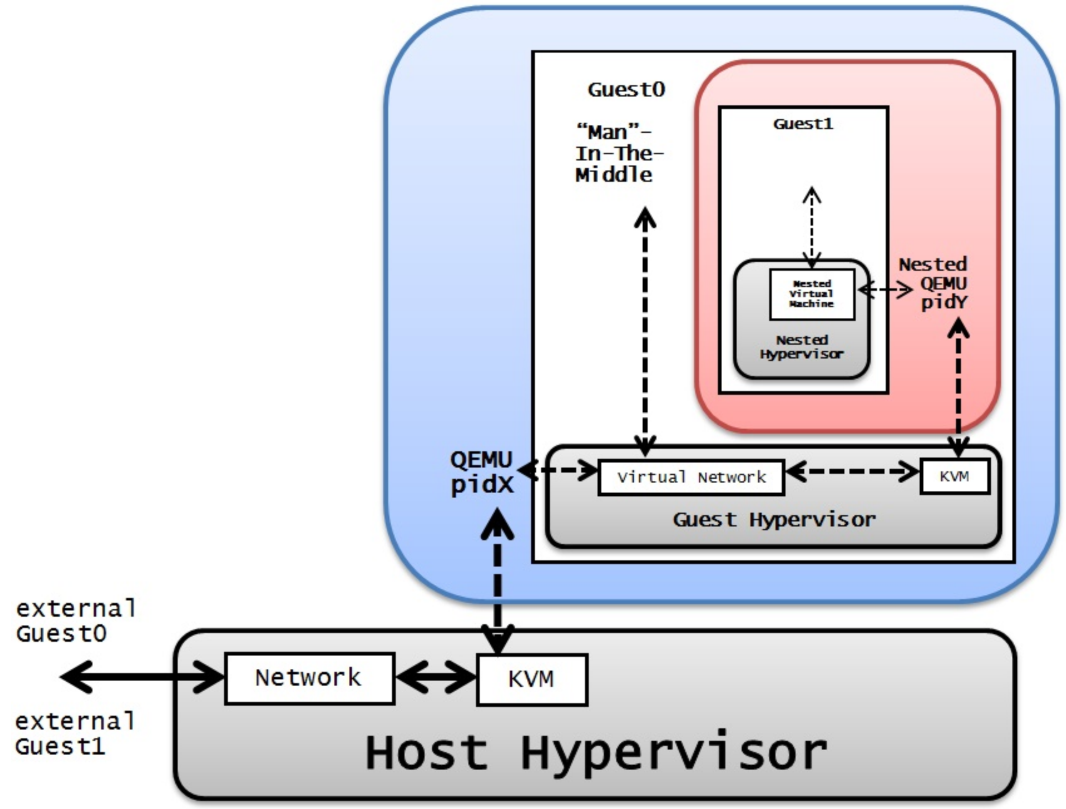

Figure 2.11: KVM/QEMU Nested Virtualization Architecture

2.2.3 Threat Model

Typically, once attackers take control of a computer system they will attempt to either hide evidence of intrusion, and/or attempt to retain that control as long as possible while carrying out some malicious service. In this work, we make a strong assumption that attackers have already compromised a victim cloud system and that our rootkit provides a stealthy environment in which the attacker can retain control of a victim guest VM owner, undetected, until the guest VM owner disconnects from the cloud.

To be clear, although compromising a cloud system is pragmatically difficult to achieve, it is possible. Gaining host privileges can be achieved by penetrating into the cloud network and exploiting certain vulnerabilities of the host machine/OS, or as we have previously provided evidence for in Section 2.2.2, VM breakout and other techniques have been demonstrated in both real world attacks and throughout the software security research community [13, 11, 50, 23, 34, 49]. The level of host privileges required after compromising the cloud system will be defined in length in Section 4.2. Under our strong assumption, attackers can create their own VMs, initiate VM live migration, and thereafter create nested VMs inside their VMs.

If an attacker's goal is to retain control of a compromised system for long periods of time, the attacker needs to hide themselves and access the system resources in an unnoticed manner. To this end, attackers usually install rootkits on the compromised system. These rootkits can help attackers to hide their malicious processes, malicious files, and malicious socket connections. Our unique type of rootkit does not seek to control or maintain the system resources of the cloud platform, instead, our goal is to gain control of the virtualized machine and services of its victim guest VM owner running on the cloud.

Based on this threat model, we will describe how attackers can leverage VM live migration and nested virtualization to create and install a CloudSkulk rootkit in Chapter 4. Again, for clarification, our rootkit allows an attacker to either actively, or passively intercept, eavesdrop, and control all QEMU/KVM data communication between a victim guest VM and its cloud platform. All live interactions of a victim guest VM owner will be visible to an attacker that has installed a CloudSkulk rootkit on some targeted IaaS based cloud environment.

Chapter 3

RELATED WORK

The foundation of our new type of software security attack is in part due to the work of many other groups of researchers. The two most influential and relevant works are two projects from 2006: SubVirt [22], then followed later that year by BluePill [39, 38]. Both detail software security attacks against a target hypervisor using a rootkit, that is represented by a VM, to maintain full control over the targeted machine. Our idea borrows from this conceptual association of a rootkit represented by a VM, but extends these concepts by adding an additional association: by exploiting live migration and nested virtualization within a cloud environment, a VM can act as the "Man" in the well known MITM type attack.

3.1 SubVirt

SubVirt was the first group to suggest this VM/rootkit association. They introduced a new type of malware which they called a Virtual Machine based Rootkit (VMBR). Their work was a proof-of-concept project in which they implemented two VMBRs to subvert Windows XP and Linux systems in a controlled lab environment. Many of the functional goals of a rootkit, that are naturally met if run as a VM, were discussed in their work and further extended in our paper. To implement their attack, the group showed that the VMBR must modify the system boot sequence of the targeted host kernel such that a VM is loaded before the target OS and applications. This was the key component of their attack, as they then prove that such a VM could then "hoist" the original OS into the VM - thus gaining full control over the target machine while remaining hard to remove and hard to detect. After installation of a VMBR the target system's disk space was fully contained within a virtual disk. However, the VMM could not translate the target's virtual disk accesses to the corresponding location on the physical disk without rebooting [22]. In other words, the target system had to be rebooted at some point to successfully install their rootkit. Figure 3.1 provides a visual representation of the SubVirt attack, showing how an original target system can be moved to run inside a VM provided by a maliciously owned and controlled VMM.

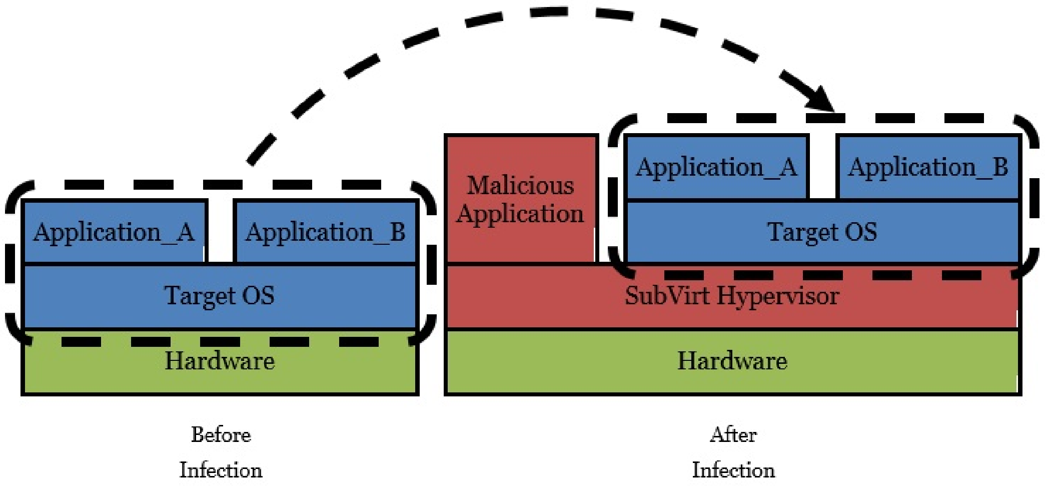

Figure 3.1: SubVirt VMBR Attack Methodology

SubVirt showed that VMBRs support general purpose malicious services. After installing a VMBR on a target system, the VMBR uses a separate OS they referred to as an "attack OS" to deploy malware. This malware was effectively invisible from the perspective of the target OS at the time because the attack OS's complete set of states and events were fully encapsulated within the attack OS and not visible from the target OS. This provides malware deployed from within the attack OS an abundant set of functionality and implementation flexibility from access to any code library, framework, OS level resource, and programming language. One exception to this was malware that executed with user-mode privileges could be detected by the target OS.

Like SubVirt, a successful installation of a CloudSkulk rootkit provides malicious attackers a powerful platform for malware. A CloudSkulk nested VM based RITM provides a malicious guest cloud user an encapsulated VM environment with many of the same benefits associated with SubVirt's VMBRs. Malware developed and deployed from within a CloudSkulk rootkit avoid detection by a host cloud platform by hiding malicious processes, malicious files, and malicious socket connections by invoking guest OS code to disallow external I/O, and protecting guest data from external introspection or modifications by checkpointing internal VM data states before and rolling the guest back later [22].

3.2 Blue Pill

The Blue Pill (BP) project, proposed and implemented by COSEINC Research and Rutkowska, J., was a software security attack that threatened operating systems without restarting the targeted system, and without any necessary modifications to the BIOS, boot sector, or system files. The main concept of the attack was that the Blue Pill represented a thin VMM that provided full control over a target OS reallocated with a VM. "The idea behind Blue Pill is simple: your operating system swallows the Blue Pill and it awakes inside the Matrix controlled by the ultra thin Blue Pill hypervisor.", project lead Rutkowska, J. [38].

The BP attack exploited AMD64 Secure Virtual Machine (SVM) assembly language extensions to move a target OS into a VM "on-the-fly". Figure 3.2 provides a visual representation of the SVM control flow for the VMRUN instruction.

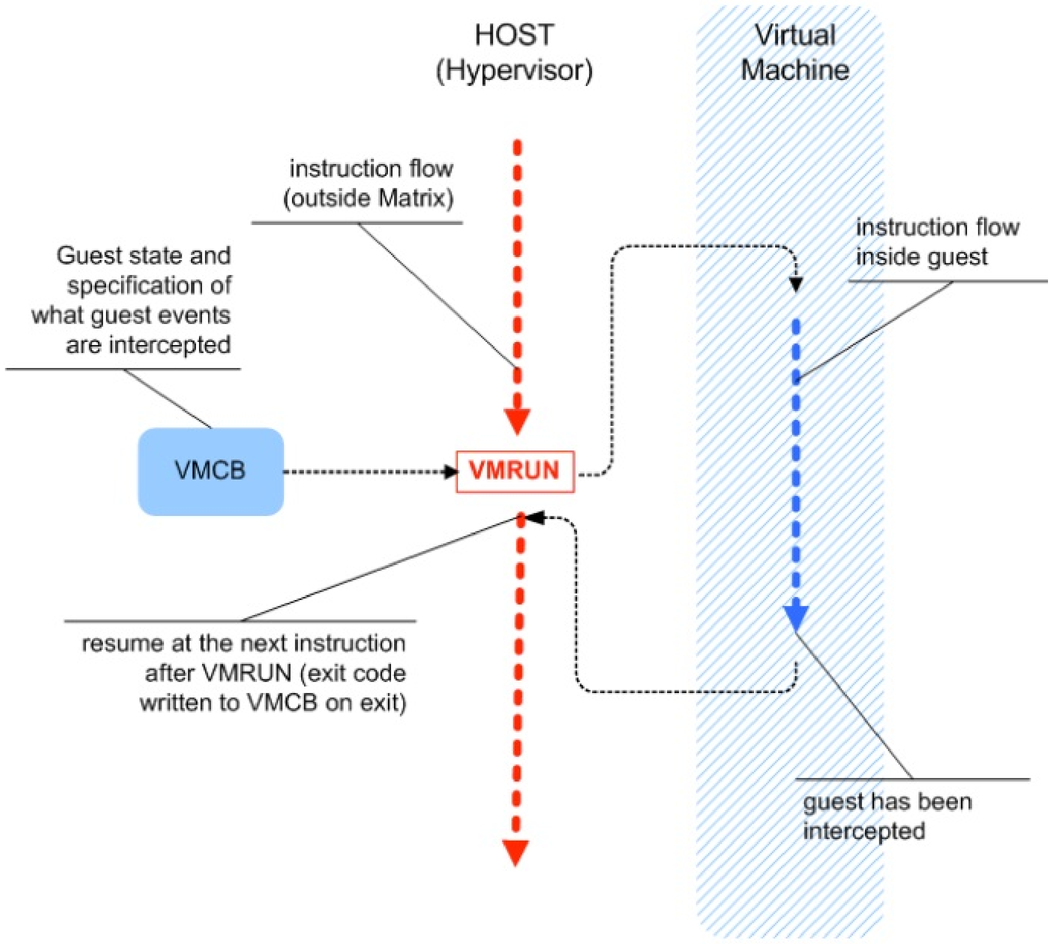
HOST (Hypervisor)
Virtual Machine
instruction flow (outside Matrix)
Guest state and specification of what guest events are intercepted
instruction flow inside guest
VMCB
VMRUN
resume at the next instruction after VMRUN (exit code written to VMCB on exit)
guest has been intercepted

Figure 3.2: Blue Pill, Heart of SVM: VMRUN Instruction [39]

The SVM instruction set contained a MSR EFER register; bit 12 of this register, the SVME bit, defined the instruction set SVM mode for the processor: 1 = enabled, or 0 = disabled. The EFER.SVME bit was required to be set high before any SVM instruction could be executed [2]. From within a host hypervisor, if the EFER.SVME bit was set to high, the VMRUN instruction could begin instruction flow from within the guest. The BP exploitation was then to modify the EFER.SVME bit during native OS execution so that a failed VMCB.exitcode check prevented the host from returning from guest mode. This SVM exploitation forced native OS execution to continue inside a VM that was fully controlled by Blue Pill. A simplified control flow diagram in Figure 3.3 depicts the Blue Pill attack.

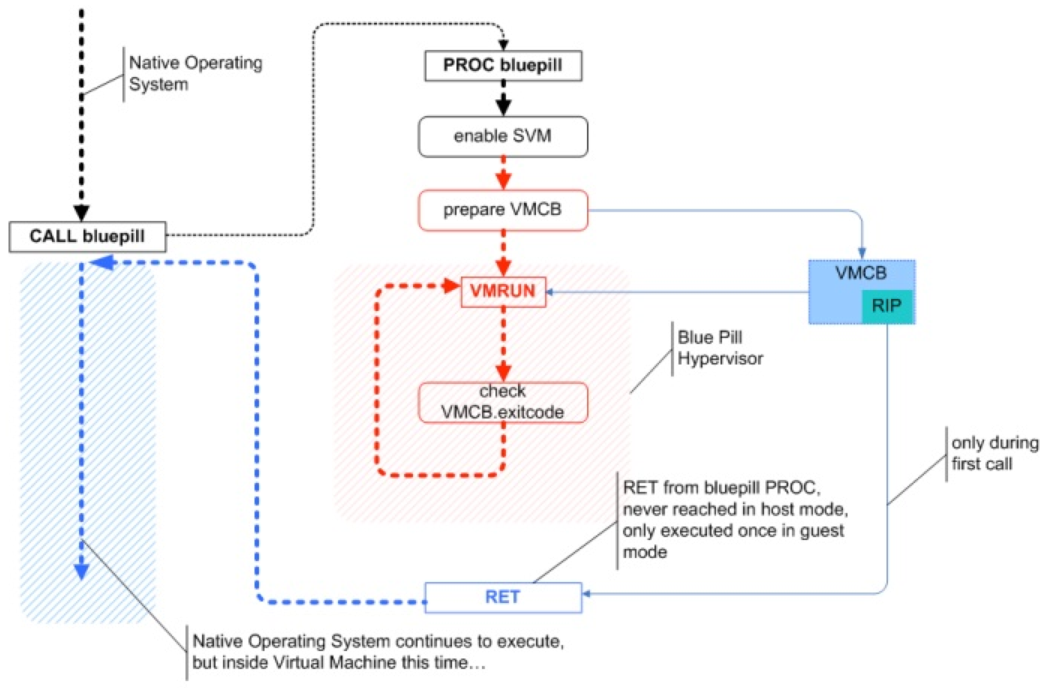
Native Operating System
PROC bluepill
enable SVM
prepare VMCB
CALL bluepill
VMCB
RIP
VMRUN
Blue Pill Hypervisor
check VMCB.exitcode
only during first call
RET from bluepill PROC, never reached in host mode, only executed once in guest mode
RET
Native Operating System continues to execute, but inside Virtual Machine this time…

Figure 3.3: Blue Pill Attack Methodology [39]

SubVirt and BluePill are kernel-hypervisor level type of attacks that are hard to implement with respect to a CloudSkulk rootkit. This is mainly because CloudSkulk requires no modifications to any host kernel or user-space code base. Our attack is also non-permanent, like Blue Pill, as we can maintain control over our nested VM only until the guest user disconnects from the cloud service provider. The most significant difference however is that SubVirt and Blue Pill were attacks against a host kernel, whereas CloudSkulk is an attack against a virtualized guest users and their host platform providing virtualization.

Chapter 4

METHODOLOGY

4.1 Design

The design of CloudSkulk is based on the Linux kernel-based virtual machine (KVM) hypervisor. In a Linux system, the KVM hypervisor is implemented as two kernel modules: one architecture independent (i.e., kvm.ko), and one architecture dependent (i.e., kvm-intel.ko or kvm-amd.ko). KVM uses hardware-level support found in modern CPU virtualization extensions, Intel VT and AMD-v, to virtualize a guest VM architecture. Each VM is then treated as a normal process, and is scheduled by the default Linux process scheduler. To create and launch VMs, users most typically employ a user-level tool called Quick Emulator (QEMU). QEMU software utilizes KVM's virtualization features to emulate an unmodified guest VM's OS, its para-virtualized and/or full-virtualized devices, and all its applications.

The rootkit we present was performed on a Linux platform that hosts the QEMU/KVM VM software paradigm. There are two key attributes that restrict our design: the virtualization software our rootkit targets must (1.) provide a utility for live migration, and (2.) enable nested hypervisors — QEMU/KVM meets these requirements. Conceptually our proposed rootkit can be installed on other cloud platforms that provide these same two attributes, but we chose QEMU/KVM specifically because of its popularity and implementation flexibility (i.e., QEMU open source code).

A CloudSkulk rootkit must be implemented in a cloud environment that provides live migration. Live migration is a common activity in the cloud; it is used to provide nearly uninterrupted service to a user while the actual server hosting services for that user can change. This activity is fundamental for cloud platforms to maintain load balance between servers, or for instances when a server requires downtime for updates or physical maintenance. Live migration does not always involve the transfer of a guest user across independent machines though, it is also used to dynamically adjust virtualized devices and VM configurations of a live user while transferring the guest on top of the same hypervisor. Our rootkit is designed to invoke live migration within a single hypervisor during its installation; we will refer to this as the VM live migration technique.

A CloudSkulk rootkit must be implemented in a virtualized environment that allows users to create nested VMs, each with their own hypervisor. This type of VM is most commonly provided, but not limited to, IaaS based cloud platforms. IaaS service providers offer users comprehensive computing resources, including user space applications, hardware, and OS, virtualized and accessed through the cloud. The design of our unique type of rootkit requires such a virtualized environment because it allows us (a malicious guest user in the cloud) to impersonate the host by running the same KVM hypervisor and QEMU VM emulator as the host. In this scenario we can then appear to mimic the host to nested guest VMs of our own; we can refer to this scenario as our nested virtualization technique. Our rootkit is called a nested VM-based Rootkit-in-the-Middle (RITM) type of attack because we can relay communication between the original host hypervisor and our nested guest hypervisor in a Man-in-the-Middle (MITM) fashion.

An installation of our unique type of rootkit is based on a comprehensive set of procedures. The following procedures define the design our attack methodology:

- Step 1: Typically in a cloud environment, an attacker, just like normal cloud customers, can rent a VM in the cloud environment. There could be many VMs co-existing on the same host machine as the attacker's VM, and one of them, would be the target for attack. In Figure 4.1, we consider GuestM to be the VM owned by the attacker, and Guest0 to be the target VM[1].

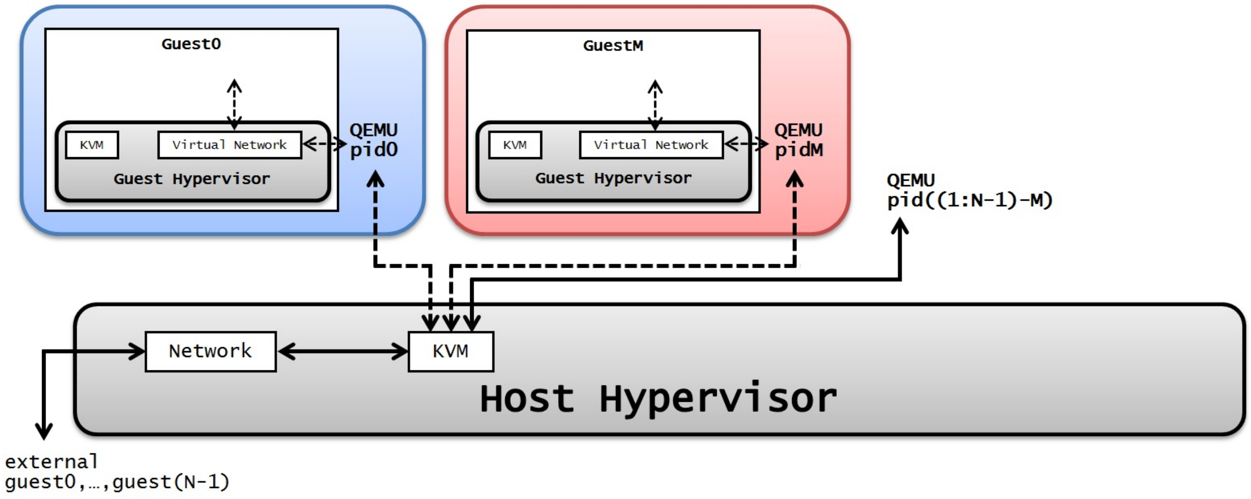

Figure 4.1: Cloud Environment Guest-Host Pair Identification

A precise definition of this step is defined as follows: There exists a set S on a GNU/Linux based host hypervisor of N number of QEMU/KVM guest-host pairs, where $|S| = N \geq 2$. In this scenario we will denote a single target of our attack as Guest0 (t), where $t \in S$, and a single attacker as GuestM (m), where $m \in S$.

[1]Figures 4.1, 4.2, 4.3, 4.4, 4.5 contain solid black arrows that denote multi-guest communication flow and dotted black arrows to denote single guest communication flow.

- Step 2: We assume that by taking advantage of existing vulnerabilities in the hypervisor, the attacker is able to break out of its VM and gain some certain control on the host [2]. This is feasible in reality as demonstrated in previous research [13, 23]. Note that the attacker does not necessarily need the system administrator privilege on the host, as a QEMU process can be launched by any normal user in a Linux system. Figure 4.2 displays a red dotted arrow that is simply an abstract representation of gaining host privileges.

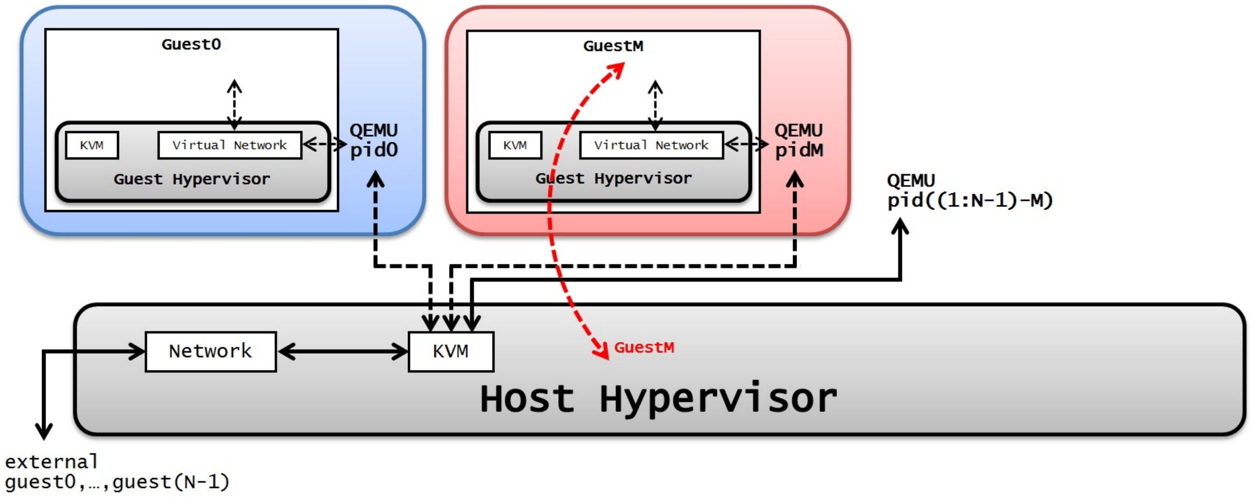

Figure 4.2: Abstraction of Virtual Machine Breakout

- Step 3: Once the attacker has some certain control on the host [2] the attacker can launch a new VM, GuestX. For the host, this VM will appear as a live migration destination of Guest0, as it will be created with all Guest0 original QEMU configurations. GuestX will functionally represent our RITM.

[2]The level of control required for this step is defined in length in Section 4.2.

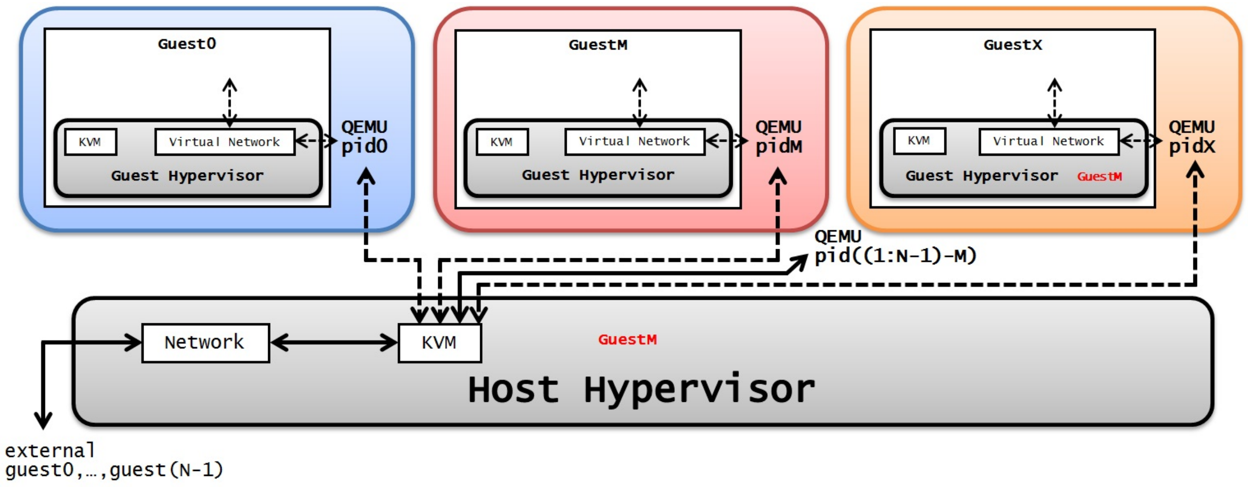

Figure 4.3: Rootkit Creation on Host

A precise definition of this step is defined as follows: An attacker, GuestM, must obtain sufficient host privileges such that the complete set of Guest0 QEMU configurations *C0* can be obtained[3]. The attacker then invokes commands on the host to create a new guest-host pair GuestX (x) with *C1* configurations, such that *C0* $\subseteq$ *C1* and where $x \cup S$, therefore $|S| = N+1 \geq 2$ and $x \in S$ are both true after this event. If *C0* was not initialized with an explicit port connection, say parameter *c'*, during its original creation, then *c'* $\cup$ *C1* must occur during this step for later live migration purposes. It is important to note that while the minimum restriction *C0* $\subseteq$ *C1* must be true, a CloudSkulk rootkit can maintain its highest level of evasion from the host if *C0* $\subseteq$ *C1* and *C1* $-$ *C0* $= \{$*c'*$\}$.

- Step 4: Utilizing the nested virtualization technique, the attacker can then launch a VM inside GuestX. This nested VM will be created with all Guest0 original QEMU configurations.

[3]The details of how this can be accomplished by an attacker, including specific commands that are most likely be used are detailed in Section 4.2.

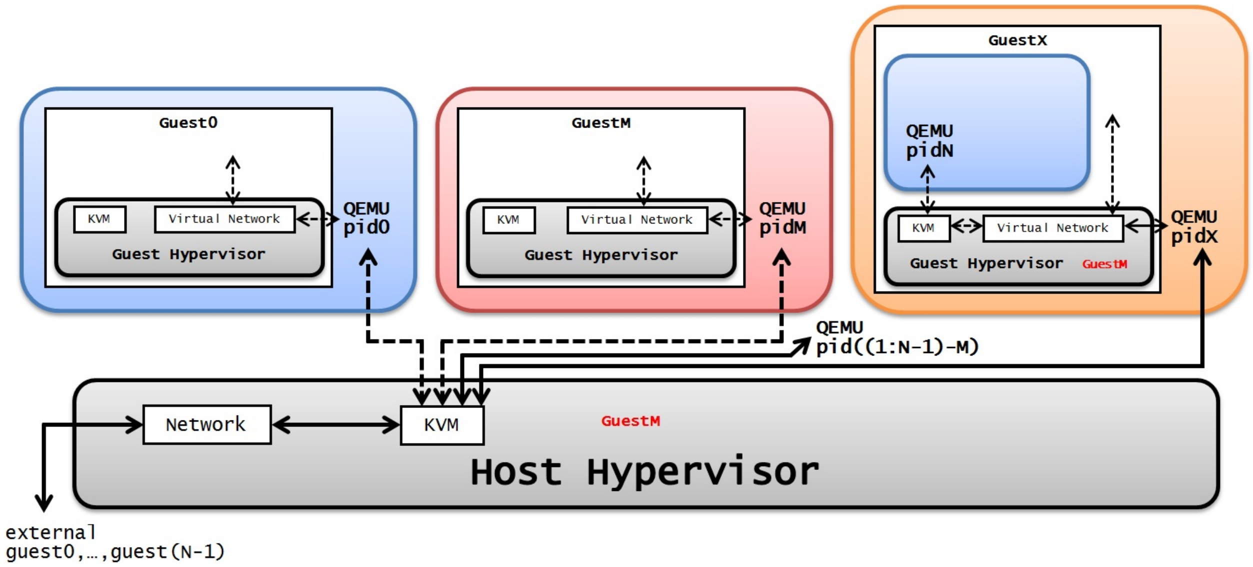

Figure 4.4: Nested VM Creation within Rootkit

A precise definition of this step is defined as follows: The attacker, GuestM, invokes commands within GuestX to create a new QEMU process[4], pidN, that is initialized in a paused state so that it contains no active user. The complete set of pidN's QEMU configurations *C2* must be intialized such that $C0 \subseteq C2$ and $C2 - C0 = \{c"\}$, where *c"* is an appended QEMU configuration that enables pidN to continuingly listen for migration data via some specified QEMU parameter[5]. The new process, pidN, is fully encapsulated within GuestX and is not visible to the host; therefore $|S|$ does not change after this event.

- Step 5: Utilizing the VM live migration technique, the attacker can migrate the target VM (Guest0) to the nested VM.

[4]The details of how this can be accomplished by an attacker are detailed in Section 4.2.
[5]A complete list of possible live migration parameters will be detailed in Section 4.2.

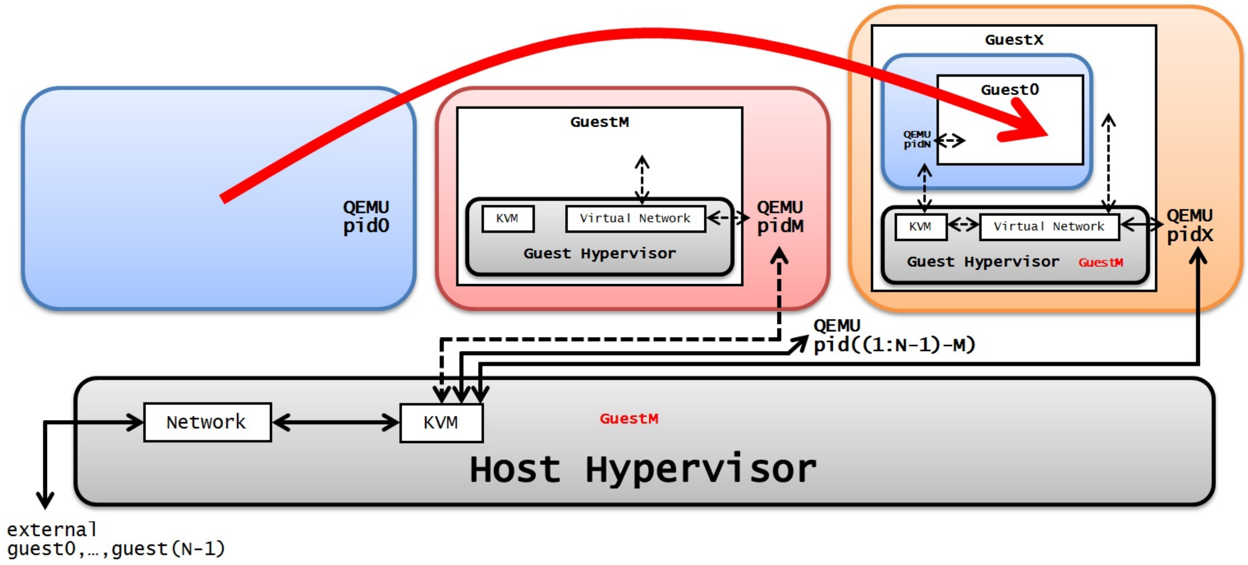

Figure 4.5: VM Live Migration to Nested VM

A precise definition of this step is defined as follows: The attacker, GuestM, invokes QEMU Monitor Console commands on the host[6] to begin live migration. The attacker can choose to migrate Guest0's active memory or disk, which when chosen is port forwarded through GuestX-host active port connection *c'*. The destination of this live migration data is then immediately sent to the nested QEMU paused process, pidN. After this step the target VM, Guest0, will be running inside GuestX as a nested VM. GuestX, our RITM, now serves as a medium that can eavesdrop on Guest0 activity and communication between the host hypervisor and Guest0 hypervisor. Figure 4.5 depicts this event, where the bold red line represents live migration of Guest0 from host's pid0 to GuestX's pidN.

[6]The exact commands required for this step are detailed in Section 4.2.

At this moment, the process pid0 (the source side of the migration) will remain on the host, but in a paused, post-migrated state with no active user. This is typical with live migration within the Cloud as a final clean-up step is required to kill the stale process. Therefore, the original condition $|S| = N \geq 2$ will apply to the host after this final event. For our installation, after an attacker has removed pid0, a nested VM-based RITM called CloudSkulk will be installed on the host.

4.1.1 Advantage of CloudSkulk

The major advantage of a CloudSkulk rootkit lies in its stealth. It is hard for both the VM owner (i.e., the victim) and the cloud system administrator to detect the existence of such a rootkit.

From the VM owner's perspective, nearly all behavioral and virtualized characteristics from within guest environment remain the same. There are several reasons, but let us first relate our rootkit installation to a typical live migration in the cloud to understand the inherent level of stealth provided by CloudSkulk. If a normal VM owner running in the cloud were to be live migrated, regardless of events independent or dependent of the guest VM, the VM owner would likely be unaware of this activity before, during, or after migration. This innate, evasive nature of live migration is by design and allows for uninterrupted services to the VM owner - a key attribute of the cloud. The guests VM machine, devices, OS, and applications are all virtualized before and after migration so various techniques of detecting virtualization cannot be applied in this scenario. For CloudSkulk, when launching Guest0 and GuestX, port forwarding is used by the attacker, so that the victim will also continue to access its VM using the same command as before. The VM owner does not observe any obvious changes except for performance. The VM owner will experience a performance change due to the additional layer of virtualization. This performance change will be characterized in Chapter 5.

From the system administrator's perspective, GuestX will now be considered as Guest0. The attacker can ensure that GuestX and Guest0 are using the same virtualized devices, the same guest OS, and run the same programs; meanwhile, with the complete control inside GuestX, the attacker has sufficient power to manipulate various Virtual Machine Introspection (VMI) techniques. This has been well documented and studied before [5]. VMI tools commonly rely on some prior knowledge of the target OS, in particular kernel-level knowledge, but when attackers are in control of the guest kernel, by manipulating various kernel data structures, attackers are able to subvert existing VMI tools. The consequence of subverting VMI tools is that attackers will be able to hide their activities or any anomaly from within the guest OS.

4.2 Implementation

Cloud environments are highly dynamic. The characteristics of a potential VM target may be configured in an assortment of QEMU image and system emulation features. In addition, specific VM parameters crucial to VM securities in cloud environments are unknown to the public. For these reasons, an attacker installing a CloudSkulk rootkit must leverage some system-level history utilities and/or VM inspection tools to expose a targeted VM configurations.

We mentioned before that some "certain" control on the host is required by the attacker who attempts to install a CloudSkulk rootkit. To clarify, the level of required control is restricted by the highest level of privilege amongst: the inspection tools needed to obtain the target VM configuration parameters, live migration, and initializing a destination VM with matching network configurations as the source VM. The level of host control will then vary depending on these restrictions.

The most straightforward solution for finding a target VM's configurations would be to investigate the command line history (*[host@cloud˜]$ history*), or report running process statuses (*[host@cloud˜]$ ps -ef*) to determine the original QEMU command used. If for whatever reason these system-level utilities are not available on the host, one powerful user-space tool, the QEMU Monitor, can be used. The QEMU Monitor is implemented alongside the QEMU source code, and is downloaded with QEMU by default. Therefore, a weak assumption can be made that this user-space utility is a common, viable solution for cloud platforms hosting QEMU/KVM. For instance, an attacker can issue a QEMU Monitor command on a running target VM to determine what block devices are emulated by QEMU (*[qemu-monitor˜]$ info qtree, [qemu-monitor˜]$ info blockstats*), or determine the size and state of the active VM memory (*[qemu-monitor˜]$ info mtree, [qemu-monitor˜]$ info mem*), or even determine the network device type, model, and state from (*[qemu-monitor˜]$ info network*). QEMU Monitor commands can also be used with other user-space utilities like qemu-img to determine the disk size of a running VM (*[qemu-monitor˜]$ info block* for the disk location, and *[host@cloud˜]$ qemu-img info* to obtain the size and type of the image file).

As a normal process of live migration in the cloud, a cloud system administrator traditionally is required to save the original configuration parameters for a user VM when it is created, or invoke a combination of any tools above to obtain these parameters during run-time. Live migration requires that this event take place because migration requires a destination VM to first be created with the same configurations as the source VM.

A CloudSkulk implementation then begins by first selecting a target VM running on the host, and obtaining its QEMU configuration parameters. An example solution to this first step is below:

```
[host@yellowstone ~]$ ps -eq | grep -i qemu
... ...
... qemu-system-x86_64 -machine type=pc-i440fx-2.3..
... ...
```

A rootkit can then be created on the host side. The rootkit is a QEMU process that matches the target QEMU parameters. Note that during testing, a destination VM could be virtualized with more devices and CPU flags than the source VM while successfully retaining attack viability. If the target VM is not originally created with a host port forwarding network configuration, we must either add this using a QEMU Monitor command (*[qemu-monitor~]$ hostfwd_add [tcp |udp]:[hostaddr]:hostport-[guestaddr]:guestport*), or we can make this during the creation of the rootkit. A continued example solution to this step is described as follows, in which we append the host port forwarding option in the creation of our rootkit:

```
qemu-system-x86_64 \
-machine type=pc-i440fx-2.3 \
-cpu Nehalem,+vmx \
-m size=512M,slots=1,maxmem=1024M \
-boot order=c \
-drive file=disk-RAW60G.img,media=disk,format=raw,
   cache=writeback,aio=threads,if=virtio \
-enable-kvm \
-vga std \
-show-cursor \
-device virtio-net-pci,netdev=net0 \
-netdev user,id=net0,hostfwd=tcp:0:4446-:5556 &
```

The guest VM environment within the rootkit is provided with its own hypervisor and the ability to nest VMs/hypervisors. Therefore, the OS within the rootkit can be modified, if needed, to enable the QEMU/KVM software infrastructure. The following provides step-by-step details for this, but is not intended as an exhaustive solution, rather a viable solution that we verify and used during implementation:

- (1.) Verify GNU/Linux x86 64 bit current OS and architecture.

  (1a.) If not, download, install, and reboot into Fedora Live Workstation x86 64 version 22.3. Then download and install the Linux Kernel version 4.4 with default configurations. Reboot and verify kvm module is enabled with new installation (*[host@cloud˜]$ lsmod |grep -i kvm*).

- (2.) Verify QEMU x86 64 bit version 2.9 is available on current machine.

  (2a.) If not, download QEMU x86 64 bit stable version 2.9, then configure and install the build with the minimum following options: *–target-list=x86_64-softmmu*, *–enable-curses*, *–enable-kvm*. Verify the correct version of qemu is enabled (*[host@cloud˜]$ qemu-system-x86_64 -version*).

- (3.) Determine if kvm_intel parameter is enabled (*[host@cloud˜]$ modinfo kvm_intel |grep -i nested*).

(3a.) If not, temporarily unload the kvm_intel module (*[host@cloud˜]$ modprobe -r kvm_intel*). Then modify the kernel module nesting parameter for kvm_intel (*[host@cloud˜]$ sudo vi /etc/modprobe.d/dist.conf*, then uncomment the line *options kvm_intel nested=y*, save and exit config file). Reboot and verify kernel changes are persistent (*[host@cloud˜]$ cat /sys/module/kvm_intel/parameters/nested*).

- (4.) Verify CPU virtualization extensions, Intel VT |AMD-v, are enabled with type = full (*[host@cloud˜]$ lscpu |grep Virtualization*).

(4a.) If not, make sure original QEMU command appends the guest's CPU extensions during creation (*[host@cloud˜]$ qemu... -cpu model,+vmx |+amd*)

After enabling the QEMU/KVM software infrastructure within the guest, the next step in implementing a CloudSkulk rootkit is the creation of the nested VM. The nested VM is the live migration destination VM with QEMU configuration parameters that match the target VM on the host side. One requirement of live migration is that the destination VM parameters be appended such that it is paused in an incoming state, and therefore it will be listening for migration data via some specified parameter. We mentioned before a complete list of possible live migration parameters would be detailed in this section; the following is that complete list as restricted by the QEMU PC system emulator options:

```
-incoming tcp:[host]:port[,to=maxport][,ipv4][,ipv6]
-incoming rdma:host:port[,ipv4][,ipv6]
   Accept incoming migration using host tcp port.
```

```
-incoming unix:socketpath
   Accept migration using host unix socket.
-incoming fd:fd
   Accept migration using host file descriptor.
-incoming exec:cmdline
   Accept migration using output specified from external command.
-incoming defer
   Accept migration using a later specified URI via QEMU monitor
   command migration_incoming.
-device ivshmem-plan,memdev=mySharedMem
   Accept migration using shared memory backend file on host.
```

The creation of the nested VM, using one of the viable QEMU live migration parameters above, is then as follows:

```
qemu-system-x86_64 \
-machine type=pc-i440fx-2.3 \
-cpu Nehalem,+vmx \
-m size=512M,slots=1,maxmem=1024M \
-boot order=c \
-drive file=copy-RAW60G.img,media=disk,format=raw,
   cache=writeback,aio=threads,if=virtio \
-enable-kvm \
-vga std \
-show-cursor \
-device virtio-net-pci,netdev=net0 \
-netdev user,id=net0 \
-incoming tcp:0:5556 &
```

Except for a minor clean-up, the final step in implementing our rootkit is to invoke the live migration utility via the QEMU Monitor. Depending on how the target VM's monitor is emulated on the host side, its QEMU Monitor can be opened in several ways. For instance, if the target VM's QEMU monitor is multiplexed onto another serial port, such as a telnet server listening on port 5555 (*[host@cloud~]$ qemu-... -serial mon:telnet:0:5555,server,nowait*), then telnet on the host side could be invoked to open the VM's QEMU Monitor (*[host@cloud~]$ telnet 0 5555*). In our continued example, since a virtualized VGA card for the VM has been created (*-vga std*), we opened the QEMU Monitor for the target VM simply with the control keystroke Ctrl+Alt+Shift+2 once the host cursor had selected the target VM. Once the QEMU Monitor is opened, the following command can be issued inside the monitor environment to invoke live migration:

```
migrate -d tcp:0:4446
```

The port numbers we chose in our example are random. However, the relationship of the port numbers with respect to the rootkit, the nested VM, and the live migration command are crucial to our implementation. The target VM begins the transaction by sending its migration data to *HOST PORT AAAA*. The rootkit was created at the host side such that it continues the transaction by forwarding *HOST PORT AAAA* to its internal *ROOTKIT PORT BBBB*. Finally, the paused nested VM will conclude the transaction as it will receive the migration data from *ROOTKIT PORT BBBB*. This transaction is visually represented by Figure 4.6.

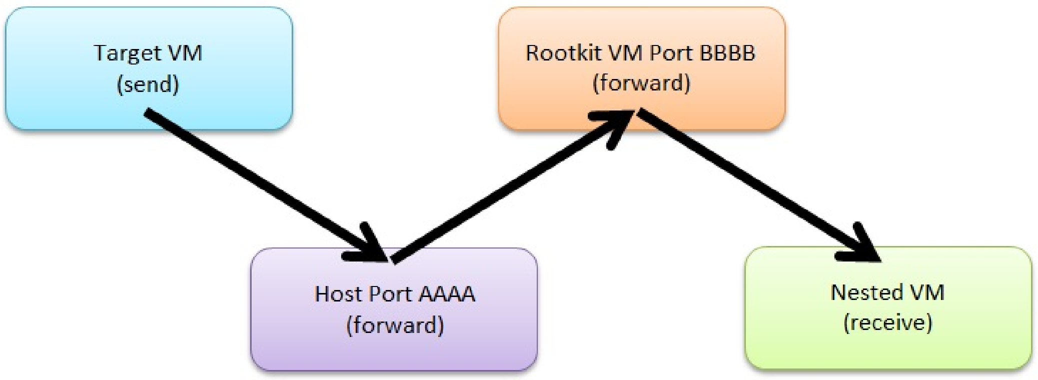

Figure 4.6: Nested Virtualization Technique Port Transaction

A minor clean-up is required after the live migration has completed. This can be accomplished by terminating the Linux process responsible for the post-migrated, paused target VM on the host side, or simply through a QEMU Monitor command still open on the post-migrated VM (*[qemu-monitor˜]$ quit*), or host-side (*[host@cloud˜]$ kill -KILL ⟨process_id⟩*). This step completes the implementation of a CloudSkulk rootkit.

4.2.1 Demonstration

To demonstrate a successful installation of CloudSkulk, we have taken a video and publicly made it available via youtube: `https://youtu.be/p4vUkADpSh4`. In the video, we assume that the attacker has already gained some control on the host system. As shown in the video, the attacker does not need a system administrator's privilege, just a normal user's privilege would suffice to perform the attack, including launching VMs and initiating VM live migration. It can be seen from the video that the time cost of the live migration is less than one minute.

Chapter 5

EVALUATION

There is no fixed threshold that defines levels of remaining unnoticed, nor is there a threshold that defines a maximum duration of installation time we are required to achieve to remain unnoticed. So instead we focus on quantitatively expressing our rootkit's ability to remain unnoticed by a target guest VM user, and the host cloud platform that our attack targets. We achieve this by characterizing both the performance degradation caused by our unique type of RITM, and its installation timing. Our goal is that this data can then be applied on a case-by-case basis to specific virtualized environments to assess the validity of our rootkit.

5.1 Performance Characterization

All the experiments are performed on a testbed running Fedora 22 operating system with GNU/Linux Kernel 4.4.14 (-200.fc22.x86_64) containing KVM. The guest Level1 and Level2 are also running the same Fedora 22 Workstation version, with Linux Kernel 4.4.14. All execution environments: Level0, Level1, and Level2 are running the latest stable version of QEMU 2.9.50 (v2.9.0-989-g43771d5) with the following install config options: *–enable-kvm*, *–enable-curses*. Our testbed platform uses Dell Precision T1700 with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz processors. The host has 16GB memory, and we assign each VM 1GB memory.

Note that QEMU offers plenty of parameters for its virtualized machines, setting different values of these parameters might affect workload performance in the VM considerably. During our experiments, we followed the QEMU/KVM best practices described by [17]. In particular, our setup has the following attributes:

- We used virtio drivers. Virtio was originally an I/O virtualization framework for Linux, but now it has also been adopted in Windows. Basically, virtio embraces the idea of para-virtualization. On the one hand, the guest OS is aware of its virtualization environment, and includes drivers that act as the front-end (these drivers need to be enabled when compiling the guest kernel); on the other hand, the hypervisor implements the back-end drivers whose major tasks are to emulate specific devices. In QEMU, when we start the VM, we need to append "*-drive if=virtio*" and "*-device virtio-net-pci*" to the QEMU command line, so as to enable the back-end disk driver and the network interface driver, respectively.

- We used block devices for VM image storage. Typically, a VM image is stored in a file on the host file system, however one can assign a device or a disk partition to the VM. In our experiments, we noticed that a block device backed VM image performs significantly better than a file backed VM. For example, for kernel compilation, the difference was as large as between 20% and 30%.

In addition, we have also experimented with different QEMU configurations. Our final results use the best case performance observed with the following parameters:

- Guest image format. The two most common image formats that are used in QEMU are the raw image format and qcow2 image format. Our best case performance and final choice was the raw image format. Our observations coincide with [17], in which raw image offers better performance than qcow2.

- Guest memory size. The start-up virtual RAM size for a guest is allocated based on this parameter. Our observations for -m size=512M, 1G, 4G, 16G, revealed 1G to be our best case performance setting.

- Guest cache mode. The three most common cache modes used in QEMU: "*cache=writeback*", "*cache=writethrough*", and "*cache=none*".

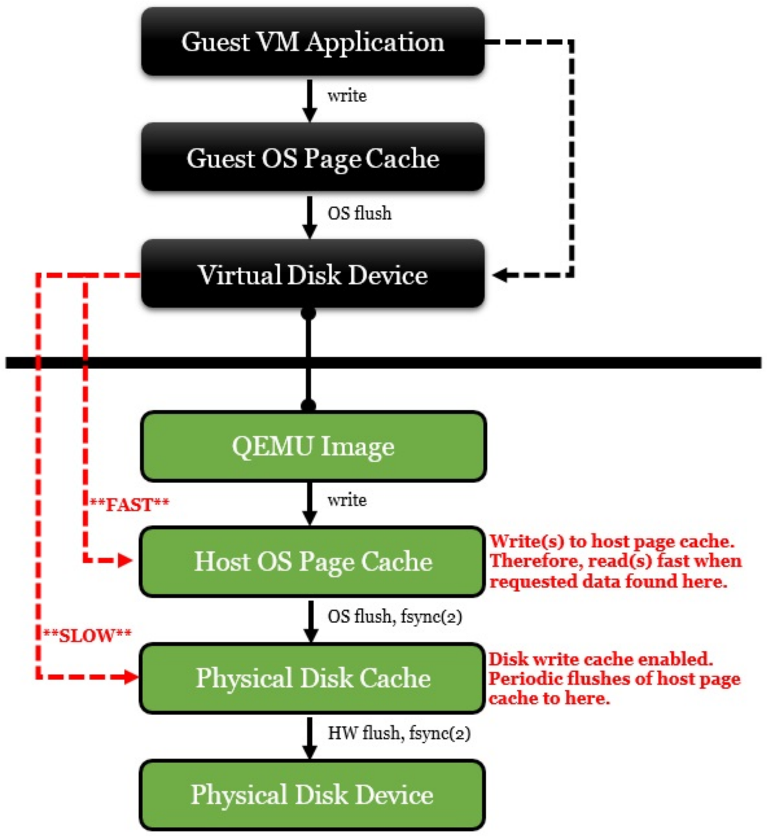

Figure 5.1: QEMU/KVM Host Cache Control Mode - "writeback"

The first mode, seen in Figure 5.1 is the default, which enables both the host OS page cache and the physical disk write cache. This caching policy will report data writes as completed as soon as the data is present in the host page cache.

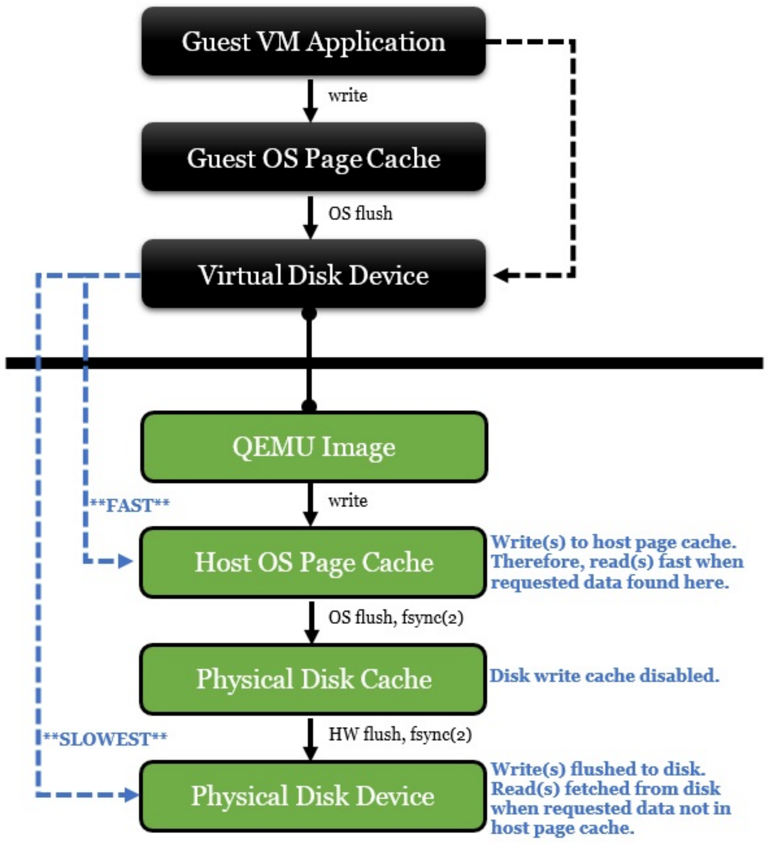
Guest VM Application
write
Guest OS Page Cache
OS flush
Virtual Disk Device
QEMU Image
FAST
write
Host OS Page Cache
Write(s) to host page cache. Therefore, read(s) fast when requested data found here.
OS flush, fsync(2)
Physical Disk Cache
Disk write cache disabled.
SLOWEST
HW flush, fsync(2)
Physical Disk Device
Write(s) flushed to disk. Read(s) fetched from disk when requested data not in host page cache.

Figure 5.2: QEMU/KVM Host Cache Control Mode - "writethrough"

In the second mode, seen in Figure 5.2, the host page cache will be used to read and write data, but the physical disk write cache is disabled. In this mode write notifications will be sent to the guest only after QEMU has made sure to flush each write to the disk. The write-through cache policy performed the worst in our experiments as each write causes a heavy IO performance impact.

The last mode, seen in Figure 5.3, disables the host OS page cache and enables the physical disk write cache. In our experiments this caching policy performed the best, slightly better than "writethrough". Our choice was cache=none.

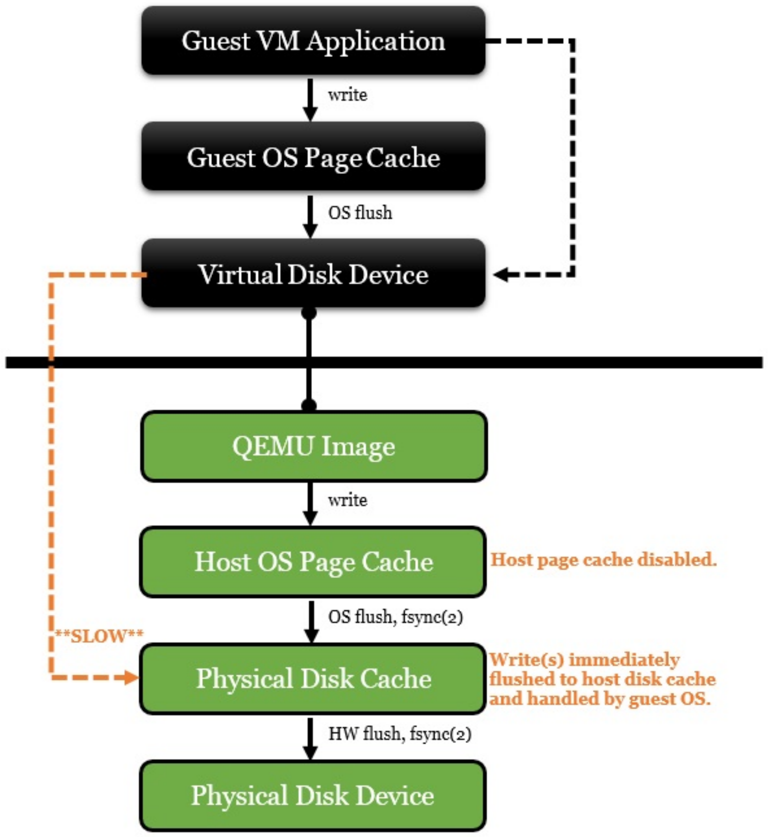

Figure 5.3: QEMU/KVM Host Cache Control Mode - "none"

- The I/O scheduler. Linux offers four different I/O schedulers: the noop scheduler, the deadline scheduler, the anticipatory scheduler, and the complete fair queuing (CFQ) scheduler. In our experiments, our choice was the default I/O scheduler, the CFQ scheduler. It was suggested [17] that using the deadline scheduler on both the host and the guest OS for I/O bound workloads should offer better performance, but we did not notice a significant performance benefit during our experiments when we switched from the CFQ scheduler to the deadline scheduler.

5.1.1 Macro Benchmarks

The performance and live migration timing of a virtual machine running in the cloud is affected by a diverse set of variables, some of which are interdependent. It is then unreasonable to exhaustively characterize these, so we follow the QEMU/KVM best practices described by [17]. For our testing strategy we choose to assume a common cloud guest user, following these above best practices, with only a single set of QEMU configuration parameters. We also know that a guest user's workload within their environment is one such variable that will play a significant role in performance and migration timing. Therefore, using an assumed cloud guest user, with static configuration parameters, we can characterize both the performance degradation and live migration timing affected by three types of workloads that summarize the generalized activity the user could be performing: IO intensive workloads, CPU/Memory intensive workloads, and Network intensive workloads.

The assumed cloud guest user's QEMU configuration parameters held constant throughout all the tests are given as follows:

Level 1 VM:

```
sudo qemu-system-x86_64 \
-name level1 \
-m size=1G,slots=1,maxmem=2G \
-boot c \
-drive file=/dev/sda6,index=0,media=disk,format=raw,
   cache=none,if=virtio \
-drive file=/dev/sda7,index=1,media=disk,format=raw,
   cache=none,if=virtio \
-cpu qemu64,+x2apic,+vmx \
-machine accel=kvm \
-smp 8 \
-curses \
-serial stdio \
-device virtio-net-pci,netdev=netLevel1 \
-netdev user,id=netLevel1,hostfwd=tcp::5022-:22
```

Level 2 VM:

```
sudo qemu-system-x86_64 \
-name level2 \
-m size=1G,slots=1,maxmem=2G \
-boot c \
-drive file=/dev/vdb,index=0,media=disk,format=raw,
   cache=none,if=virtio \
-cpu qemu64,+x2apic,+vmx \
-machine accel=kvm \
-smp 8 \
-curses \
-serial stdio \
-device virtio-net-pci,netdev=netLevel2 \
-netdev user,id=netLevel2,hostfwd=tcp::5022-:22
```

Again, as mentioned in the previous section, we used block devices for VM image storage. For testing purposes this forced us to run QEMU with super-user privileges so that we could access the host machine's block device /dev/sda6.

To evaluate I/O intensive workloads, we chose a widely used, open source benchmark called Filebench [1]. Filebench is a file level application measurement framework, written in C, which contains a set of pre-defined and configurable high-level macros. We have chosen three types of macros that we believe are the most closely associated with what I/O workloads a generalized guest user would perform: file server workloads, web server workloads, and mail server workloads.

- **The File Server macro:** consists of a combination of *createfile*, *writewholefile*, *closefile*, *openfile*, *appendfilerand*, *readwholefile*, *deletefile*, and *statfile* operations. Of all macros used, the File Server macro uniquely calls write operations.

- **The Web Server macro:** is uniquely dominated by read operations, with a combination of *openfile*, *readwholefile*, *closefile*, and *appendfilerand* operations.

- **The Mail Server macro:** uniquely calls fsync operations; the macro consists of a combination of *deletefile*, *createfile*, *appendfilerand*, *fysnc*, *closefile*, *openfile*, and *readwholefile* operations.

Table 5.1: Filebench Macro Parameters

macro	testdir	filesize	nfiles	meandirwidth	nthreads	nfilereadinstances	iosize	meanappendsize
File Server	/home	64k	50,000	20	50	1	1m	16k
Mail Server	/home	2k	50,000	1,000,000	16	1	16k	8k
Web Server	/home	16k	50,000	20	100	1	1m	16k

These three macros were configured statically with the parameters found in Table 5.1. A full code base of all custom .f files written for this testing can be found in Appendix A.1, Appendix A.2, and Appendix A.3.

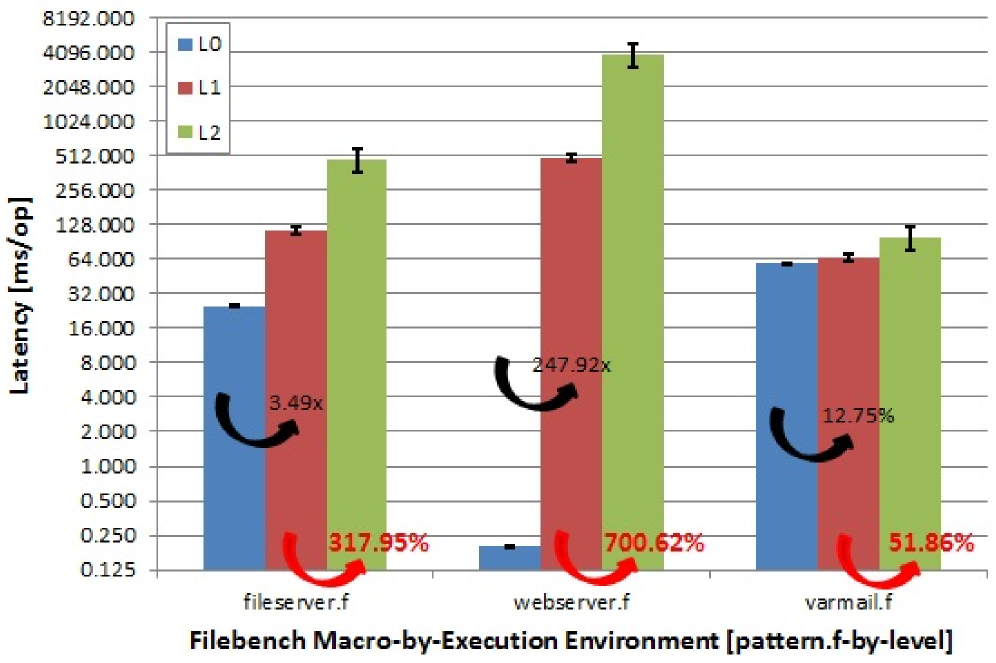

Figure 5.4: Filebench - I/O intensive workload latency analysis

The data for our Filebench tests, as shown in Figure 5.4 and Figure 5.5, were collected using a Bash shell script, found in Appendix B.1, that executed each Filebench macro five consecutive times in each execution environment and averaged the results. The L0 data series represent a Filebench workload running on the test platform that resembles our targeted cloud environment; L1 is the guest VM environment, and L2 is the nested VM environment. The x-axis in both figures displays the three L0:L2 execution environments, each against the three .f macros tested, for nine total data series. The y-axis in both figures displays the filebench results in log base 10 scale. The data labels [1] in Figure 5.4 show the percentage increase in latency with respect to the layer below it. The data labels in Figure 5.5 show the percentage decrease in throughput with respect to the layer below it. Each data series displays its relative standard deviation in a bar centered on the top of each column.

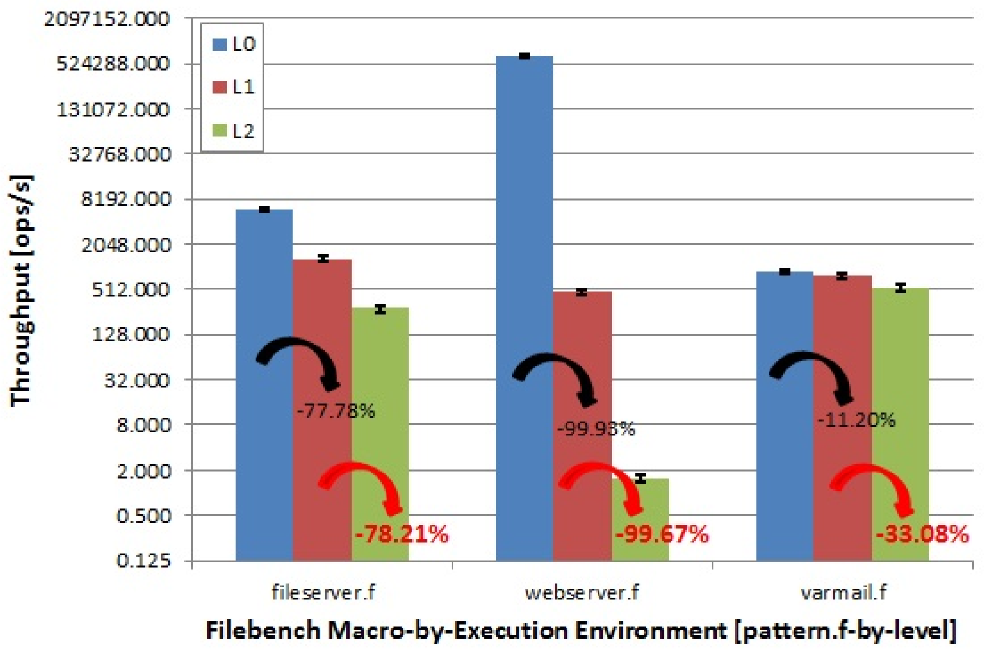

Figure 5.5: Filebench - I/O intensive workload throughput analysis

By evaluating the percent difference of the latency and throughput between L1 and L2 in Figure 5.4 and Figure 5.5 we can quantitatively express what a guest user's perspective is before and after the installation of a CloudSkulk rootkit for IO intensive workloads. After our rootkit is installation, a targeted guest user will

[1]Figure 5.4 - Figure 5.8 display L1 data labels that are centered in each L1 data series bar. L2 data labels are at the inside base in each L2 data series bar.

experience a 51.86% decrease in speed and 33.08% decrease in throughput for Mail Server type workloads, ~3.18x slower speeds and 99.67% decrease in throughput for File Server type IO workloads, and ~7.01x slower speeds and 78.21% decrease in throughput for Web Server type IO workloads. Nested KVM results in literature showed a ~46% best case and ~67% worst case throughput decrease for similar Filebench IO intensive testing [26]. While our IO throughput decrease results coincide with others, our results collectively are still poor. Most importantly, of all our test data shared in this paper our poor IO latency results do not correlate with literature results; whereas similar nested KVM literature data showed a best case of ~10:30% increase in IO latency [26], our results show that a CloudSkulk compromised guest user of the cloud will almost certainly be aware of the heavy performance degradation if they are actively performing IO intensive workloads. This drawback makes the current version of our rootkit prone to cloud security detection systems.

To evaluate our second focus on CPU/Memory intensive workloads, we chose to collect Linux Kernel decompression and compile times for the same three execution environments as our previous benchmark: L0, L1, and L2. By nature, the kernel compile process is CPU intensive and Memory intensive. Again, we wrote a Bash shell script, found in Appendix B.2, sharing the exact same .config file created on L0 for all tests, and decompressed, then compiled the Linux Kernel version 4.0 five consecutive times and averaged the results. This characterization data can be seen in Figure 5.6 and Figure 5.7. The x-axis in both figures displays three data series, one for each L0:L2 execution environment. The y-axis in both figures displays the timing results in log base 10 scale. The data labels in both figures show the percentage increase in timing with respect to the layer below it. Each data series displays its relative standard deviation in a bar centered on the top of each column.

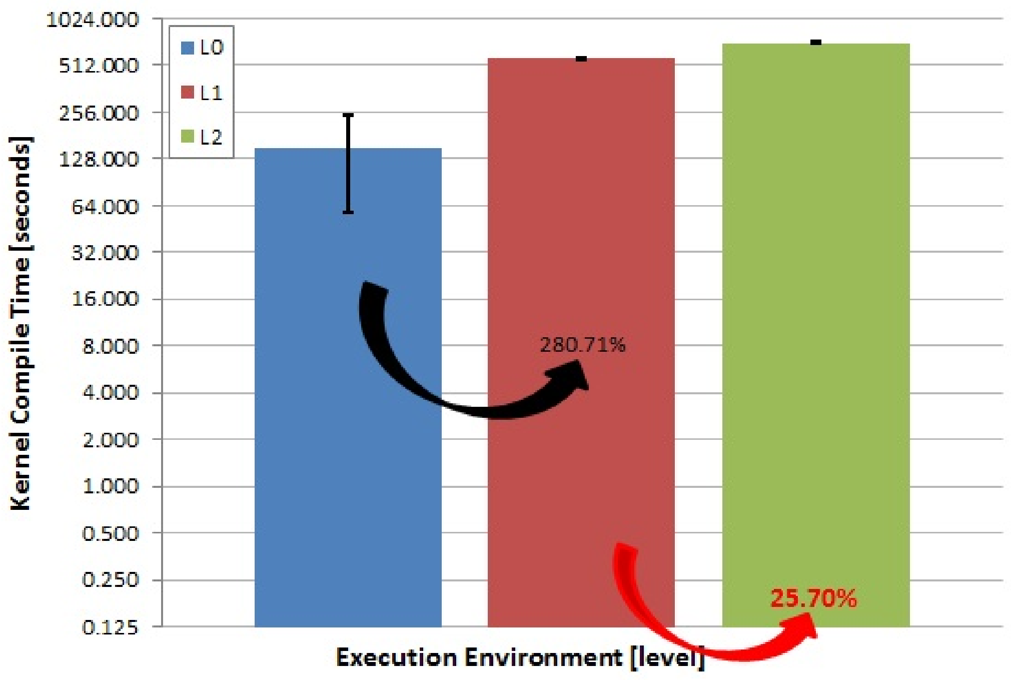

Figure 5.6: Kernel Compile - CPU/Memory intensive workload analysis

As before, we can quantitatively express the performance degradation perceived by a guest user for CPU/Memory intensive workloads before and after the installation of a CloudSkulk rootkit by evaluating the percentage difference of Kernel Decompression and compile time between L1 and L2 in Figure 5.6 and Figure 5.7. After our rootkit is installed, a targeted guest user will experience a 25.7% decrease in speed associated with the Kernel-compile type of CPU/Memory workloads. This coincides with Intel performance evaluation results of KVM in 2014 [12] of ∼26.31% decrease from L1 to L2 virtualization: $L2\_time = 14s, L1\_time = 19s, \therefore L2\_impact = 100\% * (19s - 14s)/19s$. This data was taken using the same number of vCPUS used in our testing ("-smp 8") for their Kernel Compilation testing. It is interesting to note that that the relative standard deviation for L0 Kernel compile time was very high (a value of 92.388% not displayed in the figure). This was due to the first run being ∼3x slower than its last 4 consecutive runs. This kernel compile data point was repeatable.

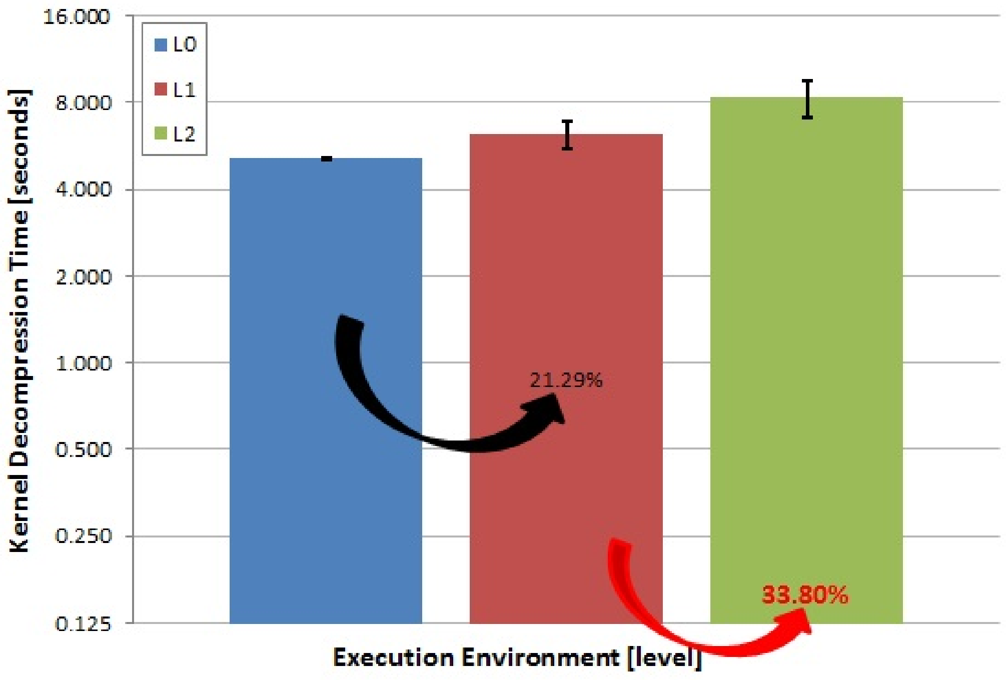

Figure 5.7: Kernel Decompression - CPU/Memory intensive workload analysis

For our third performance focus, we chose to use another well-known, widely used open source benchmark, Netperf [19]. Netperf is a network performance benchmark, written in C, which is used to measure networking performance based on bulk data transfer and request/response performance using the TCP/UDP network protocols. For our testing, we chose to measure the bulk data transfer performance, or uni-directional stream performance of TCP. We wrote a Bash shell script, found in Appendix B.3, that executed the Netperf application five consecutive times and averaged the results for the same L0:L2 execution environments. Our Netperf benchmark data can be seen in Figure 5.8. The data labels in this figure show the percentage decrease in latency with respect to the layer below it. The x-axis in this figure displays three data series, one for each L0:L2 execution environment. The y-axis in this figure displays the throughput results in log base 10 scale. Each data series displays its relative standard deviation in a bar centered on the top of each column.

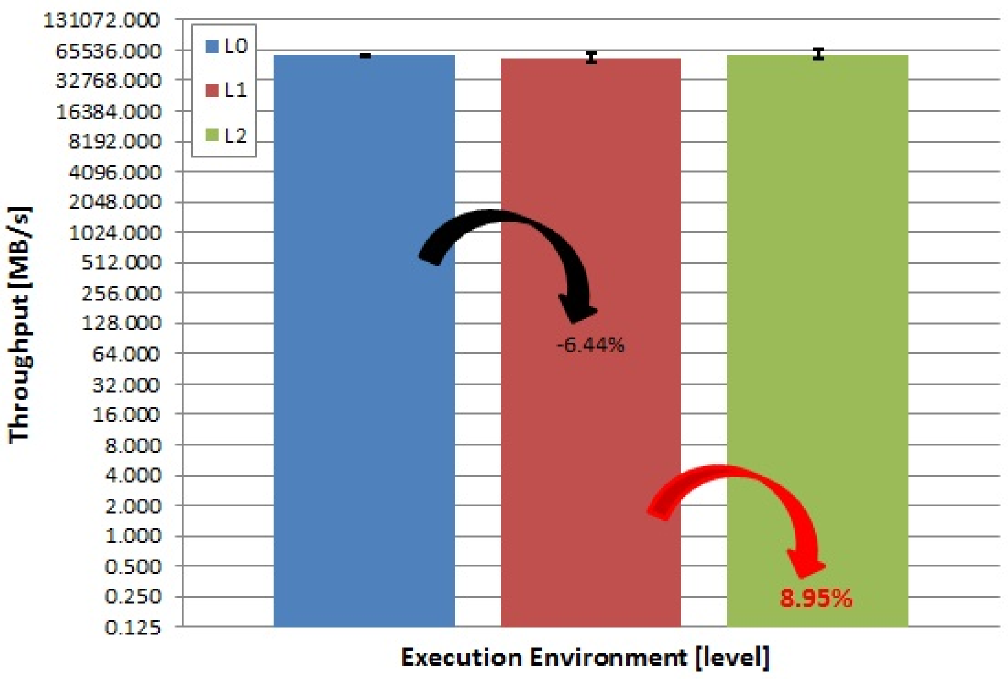

Figure 5.8: Netperf - Network intensive workload throughput analysis

Again, the percentage difference of the throughput between L1 and L2 in Figure 5.8 quantitatively show the performance degradation perceived by a guest user for network intensive workloads before and after the installation of a CloudSkulk rootkit. As visually eluded to by the relative standard deviation bars, all three levels of the execution environment performed nearly the same with overlapping data sets. The averages show a 8.95% increase in throughput for the TCP bulk data transfer type of network workloads after our rootkit installation, with standard deviations (explicit values not shown in figure) for L0:L2 being 1.11%, 10.32%, and 3.96%, respectively. With standard deviations higher than the percentage differences in throughput, we can conclude that this performance was nearly the same across all execution environments. These results demonstrate that a CloudSkulk compromised guest user of the cloud will be unable to detect a performance decrease while performing network intensive workloads.

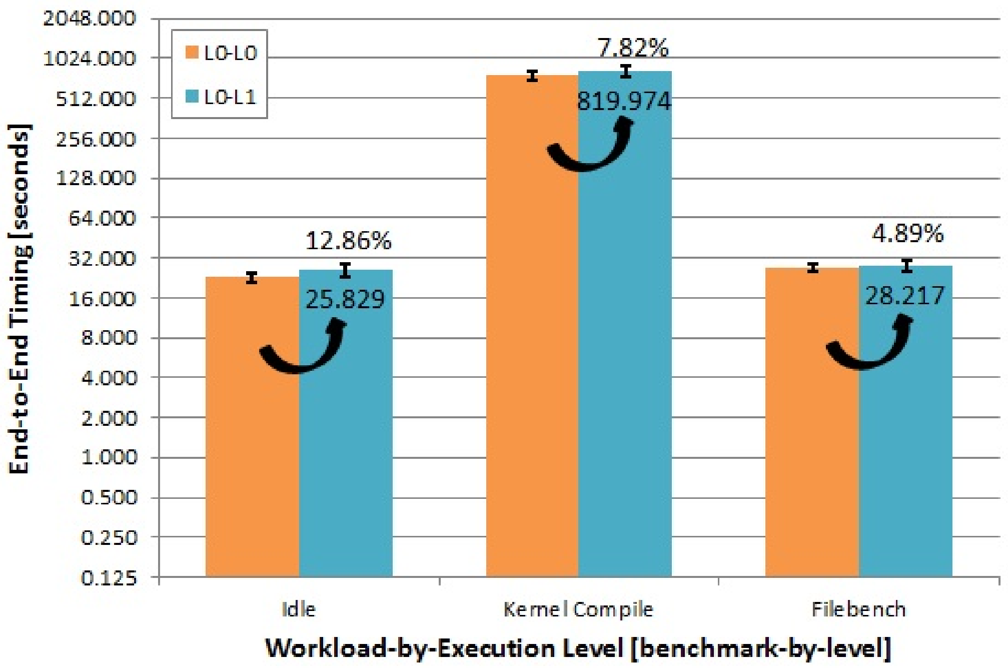

Figure 5.9: Live Migration - end-to-end timing analysis

For our last evaluation, we chose to characterize the live migration timing of a guest user performing various types of workloads: idle, Kernel compile, and Filebench.

An idle workload is represented by a guest user that is not executing any workload - this can be thought of a user connected to the cloud, but away from their device or inactive. For this testing, we chose two levels of live migrations to characterize: L0-L0, and L0-L1. This live migration characterization data can be seen in Figure 5.9.

The L0-L0 data series in Figure 5.9 depict a typical invocation of live migration in the cloud, except that there is no network traffic generated during the migration [2], because both the source and destination VMs coexist on a single, commonly shared environment. The L0-L1 data series in Figure 5.9 depict our unique nested VM-based technique used to implement CloudSkulk. This type of migration involves a L1 VM running on the host side to be live migrated into an L2 nested VM that is encapsulated within our L1 rootkit VM (VM-based RITM). The common metric for live migration is the total migration time (end-to-end time). Each data point in Figure 5.9 is the average end-to-end time for 5 consecutive runs with their corresponding relative standard deviation displayed in bars centered above each column.

One important note to make was that L1 and L2 VM configurations for live migration testing was slightly different than the other characterization data discussed above. For consistency, the following details the difference from previous parameters to live migration parameters:

VM Live Migration Config. Differences:

```
from: -m size=1G,slots=1,maxmem=2G \
to:   -m size=512M,slots=1,maxmem=1024M \
```

[2]This means in a real cloud environment, the live migration time could be considerably longer than those appear in Figure 5.9, because of the large volume of network traffic, especially when disk migration is involved.

```
from: -drive file=/dev/<disk>,index=0,media=disk,
        format=raw,cache=none,if=virtio \
to:   -drive file=/tmp/guest-imgs/copyRAW60G.img,
        media=disk,format=raw,cache=writeback,aio=threads,if=virtio \

from: -cpu qemu64,+x2apic,+vmx \
to:   -cpu Nehalem \

from: -machine accel=kvm \
to:   -enable-kvm \

new:  -machine type=pc-i440fx-2.3 \
```

There are two sets of data labels in Figure 5.9. The bottom-most set of data labels show the numerical values associated with each end-to-end time, and the top-most set shows the percentage increase in end-to-end time from L0-L0 to L0-L1. The bottom-most set of data labels are important and more directly relevant to our evaluation process. These values allow us to determine our CloudSkulk installation time based on what workload activities the user could be performing. Since a CloudSkulk installation time is dominated almost entirely by the time based on the nested live migration step, we will approximate the total installation time below by referring to it as the integer ceiling of its nested live migration end-to-end time. Therefore, the best case installation time of a CloudSkulk rootkit is ∼26 seconds. This occurs when the target guest workload is idle. The installation time of CloudSkulk when a target guest user is performing I/O intensive workloads is ∼29 seconds; for

CPU/Memory intensive workloads, the time is ∼820 seconds. The top-most set of data in Figure 5.9 is less relevant, but still important to note. This data labels show us how much extra time will be added to our live migration end-to-end time with respect to the nominal L0-L0 type migrations.

A final important note to make is that the live migration downtime was recorded for each data point. However, this data was not displayed for two main reasons: across all execution environments, the standard deviation was high (∼50-60%), and no single downtime was above 103ms, which would arguably be unnoticeable by a guest user.

5.1.2 Micro Benchmarks

To more precisely measure the overheads, we performed a number of microbenchmark tests. We chose lmbench version 3.0-a9 as our microbenchmark [31]. Our experimental results are presented in Table 5.2, Table 5.3, Table 5.4, and Table 5.5.

Table 5.2: lmbench: Arithmetic operations - times in nanoseconds

Config	integer bit	integer add	integer div	integer mod	float add	float mul	float div	double add	double mul	double div
Level0	0.26	0.13	5.94	6.37	0.75	1.25	3.31	0.75	1.25	5.06
Level1	0.25	0.13	5.96	6.39	0.75	1.26	3.32	0.75	1.26	5.07
Level2	0.26	0.13	6.14	6.59	0.78	1.30	3.43	0.78	1.30	5.23

Table 5.3: lmbench: Processes - times in microseconds

Config	signal handler installation	signal handler overhead	protection fault	pipe latency	AF_UNIX sock stream latency	fork+exit	fork+execve	fork+/bin/sh -c
Level0	0.075	0.50	0.27	3.49	3.58	74.6	245.8	918.7
Level1	0.096	0.58	0.29	6.75	5.37	73.65	275.05	966.67
Level2	0.10	0.60	0.32	65.49	43.98	242.19	588.50	1826.00

It can be seen from these four tables, virtualization (including nested virtualization) has negligible effect on all arithmetic operations. Also, for file creation and deletion operations, both Level 2 performance and Level 1 performance match the baseline, i.e., the Level 0 performance. For context switches, Level 1 incurs a dramatic performance degradation, and Level 2 further incurs a even more performance degradation. In addition, process fork generates big performance overhead in Level 2, likely because of the extra traps into the Level 0 hypervisor [48].

Table 5.4: lmbench: File system latency - files creations/deletions per second - times in microseconds

File Size / Level	File Size 0K		File Size 1K		File Size 4K		File Size 10K	
	File Creation	File Deletion	File Creation	File Deletion	File Creation	File Deletion	File Creation	File Deletion
Level0	126,418	379,158	99,112	280,884	99,627	279,893	79,869	214,767
Level1	121,718	361,860	97,073	268,977	95,821	273,863	77,118	204,260
Level2	2,430	320,349	62,933	262,478	96,588	251,766	70,098	196,449

Table 5.5: lmbench: Context switch - times in microseconds

Config	2p 0K	2p 16K	2p 64K	4p 0K	4p 16K	4p 64K	8p 0K	8p 16K	8p 64K	16p 0K	16p 16K	16p 64K
Level0	1.43	1.71	1.91	1.67	1.84	1.90	1.84	1.94	2.06	1.88	2.03	2.34
Level1	3.09	3.10	3.52	1.78	3.43	2.84	2.73	3.24	1.96	3.72	5.35	4.21
Level2	19.02	28.21	28.39	10.28	34.70	53.94	5.99	32.42	24.49	6.42	39.53	4.41

5.2 Defending against CloudSkulk

Rootkits installed on a compromised system allow attackers the ability to hide themselves and access system resources in an unnoticed manner. Our rootkit, CloudSkulk, allows an attacker in a cloud environment to gain control of the virtualized machine and services of a victim guest VM owner running in the cloud while remaining undetected. Our unique type of rootkit is hard to detect because its installation uses common tools and functions found in typical operating cloud environments, and its system effect after installation is minimal. Nonetheless, a CloudSkulk rootkit does leave signs of its presence that we will soon discuss. For cloud environments that do not seek to monitor these signs, the impact of a CloudSkulk rootkit may be quite high. However, as it is the goal of our research to advance cloud system securities against such attacks, cloud environments that implement monitors to detect these signs can mitigate the impact of nested VM based RITM rootkit attacks.

We can categorize various techniques that can be used to prevent and detect the presence of a CloudSkulk rootkit by the following groups: (1.) Performance Detectors, (2.) Restricting Commands, (3.) Memory Analysis Tools.

5.2.1 Performance Detectors

The installation of a nested VM based RITM rootkit intrinsically causes a performance degradation on the victim guest VM owner running in the cloud. This is due to guest running on top of an additional layer of virtualization after the attack, from L1 to L2. This performance degradation can be slightly mitigated when the guest is running para-virtualized devices, like the virtio drivers detailed in Section 5.1, but can not be fully minimized. Therefore, a nested VM based RITM rootkit may be detected by monitoring performance anomalies imposed on a guest VM after live migration.

Despite some of the high L2 IO performance impacts we presented in Section 5.1.1, the performance anomalies caused by our rootkit should coincide with all other standard QEMU/KVM nested virtualization. Cloud security techniques attempting to detect our rootkit through the performance anomalies caused by nested virtualization is however a weak strategy. This is two part: (1.) Each VM is highly dynamic, so determining performance thresholds may be difficult and unreliable, (2.) Also, as nested virtualization popularity continues to grow it is likely its performance impact will continue to decrease.

5.2.2 Restricting Commands

Cloud security software may impose restrictions on host commands to prevent nested VM based RITM attacks. By both monitoring and restricting live migration commands, system tools that can be used to find targeted guest VM configurations, and other system tools used during installation, a CloudSkulk rootkit may be blocked entirely. This technique, although viable, is significantly weaker than the previously discussed strategies because the set of commands and tools that an attacker can use during rootkit installation is difficult to quantize [3].

For instance, Step 3 in the installation of a CloudSkulk rootkit [4], the attacker is required to find the QEMU virtualization configurations of it's target guest VM. The number of ways an attacker can obtain this data is hard to approximate or define, and therefore is a weak strategy towards preventing any such commands. Another weakness to this strategy is that preventing commands like the QEMU Monitor set significantly decreases the host's functional ability to inspect, adjust, and control various aspects of it's guest virtual machines. Also, if cloud security software imposes privilege restrictions on live migration commands, the level of control the attacker is

[3]Various system tools and other solutions for finding a target VM's virtualized configurations, as well as a complete set of live migration commands for QEMU are detailed in Section 4.2.

[4]The complete design and description of each step of a CloudSkulk rootkit installation is described in Section 4.1.

required to obtain is simply lower than the restriction - at lowest ring 0 privileges. This may make the installation of a CloudSkulk rootkit more pragmatically difficult for an attacker, but does not fully prevent the attack and is therefore also a weak strategy.

5.2.3 Memory Analysis Tools

Of all techniques we have presented, memory analysis tools may provide the most reliable and strongest way to detect the presence of a nested VM based RITM. Regardless of the number of layers of virtualization within a targeted guest VM, after installation of a CloudSkulk rootkit that number will increase by one. Unless modifications (currently unknown) to the QEMU virtualization code base are implemented within the L1 (rootkit) environment, a CloudSkulk rootkit currently can not be created without imposing this memory anomaly on it's targeted host cloud platform.

The first design and implementation a memory forensic framework to analyze and recognize hypervisors, nested or single, was developed by Graziano, M. and a team of researchers at Eurecom, France in 2013 [16]. The team developed a hypervisor memory forensics tool that analyzed hypervisor structures found in physical memory dumps. Their tool successfully recognized several open source and commercial nested hypervisors installed with various configurations. Memory forensics tools like the one developed by Graziano, M. may be used by cloud security detection software to detect nested VM based RITM attacks. Such detection software may be able to use memory forensic tools to determine the layers of virtualization, specifically the number of nested hypervisors, before and after any set of VM dependent live migration commands are invoked on the host. The presence of a CloudSkulk rootkit can be detected if this number of virtualization layers increases. If a pre-copy based VM live migration strategy is deployed by the cloud, as with QEMU/KVM, detecting this hypervisor anomaly can be done by comparing the number of nested hypervisors immediately before the Iterative Copy stage and immediate after Activation stage.

It is also important to note, although not investigated, that solutions for either the attacker or defender many be implemented within the QEMU code base. Detecting or preventing the detection of the number of layers of virtualization before and after live migration may be appended within QEMU software to attempt to, again, either detect or prevent this type of attack. Another possible QEMU software solution could be to outright disallow the creation of additional layers of hypervisors, applied immediately before, during, and immediately after live migration.

Chapter 6

CONCLUSIONS

Contemporary rootkits today share a common weakness, their presence is generally detectable by software security defenders that run at a lower-level than the installed rootkit. In this thesis, we have identified a solution to this weakness. To our knowledge we are the first to reveal and demonstrate that nested virtualization can be used by attackers for developing malicious rootkits that are hard to detect, regardless of whether defenders are at a lower-level or at a higher-level than the installed rootkit. With clear evidence of virtualized cloud computing services quickly accelerating – market research forecasts an increase in cloud workloads by more than triple, 3.3-fold, from 2014-2019 [33] – the security of those working in the cloud and of their data are becoming increasingly relevant. In this thesis, we presented a unique type of nested Virtual Machine (VM) based Rootkit-in-the-Middle (RITM), called CloudSkulk, that can be used to help attackers seize and hide their control of a targeted guest VM owner running in a cloud environment. We assume the role of an attacker; by taking an offensive, malicious approach at targeting such cloud environments, it is the goal of our research then to increase cloud security against such attacks by identifying and providing possible solutions to this new type of invasive rootkit.

A CloudSkulk rootkit does not seek to access or control the cloud platform system resources, instead it seeks to seize and maintain control of a single victim (a guest

VM owner) by passively relaying all QEMU/KVM virutalization data between the host cloud platform and the guest in a MITM fashion. Although our CloudSkulk design, implementation, and demonstration is on the widely popular QEMU/KVM virtualizaiton software, our new type of rootkit can be applied orthogonally to other hypervisors supporting the two minimum implementation attributes: the virtualization software must (1.) provide a utility for live migration, and (2.) enable nested hypervisors.

Despite nested VM based RITMs providing more stealth for attackers on the cloud than contemporary rootkits, our unique type of rootkit does leave signs of its presence. For cloud environments that do not seek to monitor these signs, the impact of a CloudSkulk rootkit may be quite high. However, we explored three categories of possible techniques that can be used to prevent and detect the presence of a CloudSkulk rootkit: (1.) performance detectors, (2.) restricting commands, and (3.) memory analysis tools. Because CloudSkulk is a new type of attack and therefore unlikely that these defense measures are currently implemented, the only noticeable signs of our rootkit should be perspective differences in performance for the victim guest user. This perspective difference is unavoidable by design of our rootkit and is caused by the additional layer of virtualization imposed on the compromised guest.

We characterized this performance difference as to quantitatively express our rootkit's ability to remain unnoticed by cloud environments not actively monitoring for other detection signs listed above. Our results showed that a guest VM owner would be unlikely to notice our rootkit's impact while performing network intensive workloads, as their performance was nearly the same before and after being compromised. Our results for CPU/Memory intensive workloads show that a guest VM owner will experience a 25.7% decrease in speed after an installation of our rootkit.

These results also coincides with Intel performance evaluation results [12]. While our IO throughput data also correlated with literature results [26], our results collectively are poor. In fact, we showed that under our current implementation, a CloudSkulk rootkit's impact on performance would likely perceived by a compromised guest user performing IO intensive workloads.

Despite these drawbacks, the best case installation time for our rootkit is $\sim$26 seconds. With fast installation times, relatively low implementation costs, and the unique additional stealth provided for attackers, we believe that a nested VM based RITM attack is a viable treat that is impactful for cloud environments who do not seek to detect it's unique presence.

References

[1] Filebench. `https://github.com/filebench`. [Online; accessed 4-June-2017].

[2] Inc Advanced Micro Devices. *Secure Virtual Machine Architecture Reference Manual*. AMD.

[3] Sanjay P. Ahuja. Full and para virtualization, 2014.

[4] Ahmed M Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*, pages 38–49. ACM, 2010.

[5] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. DKSM: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 82–91. IEEE, 2010.

[6] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har&#039;El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, 2010. USENIX Association.

[7] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 409–417. IEEE, 2001.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation (NSDI) - Volume 2*, pages 273–286. USENIX Association, 2005.

[9] CloudPassage. Share the cloud security spotlight report, 2016.

[10] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE, 2000.

[11] CVEDetails.com. Vulnerability details : CVE-2007-1744, 2007.

[12] Bandan Das, Yang Zhang, and Jan Kiszka. Nested virtualization, state of the art and future directions, 2014.

[13] Nelson Elhage. Virtunoid: Breaking out of KVM. *Black Hat USA*, 2011.

[14] Ans. Editor Gard Steiro and Managing. Director. Great overview: Services for mobile payment. 2016.

[15] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, volume 3, pages 191–206, 2003.

[16] Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. Hypervisor memory forensics. In *International Workshop on Recent Advances in Intrusion Detection*, pages 21–40. Springer, 2013.

[17] IBM. Kernel virtual machine (KVM) - best practices for KVM. `https://www.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaatbestpractices_pdf.pdf`, 2012. [Online; accessed 1-June-2017].

[18] IntegrantSoftware. IaaS vs. PaaS vs. SaaS, 2013.

[19] Rick Jones et al. NetPerf: a network performance benchmark. *Information Networks Division, Hewlett-Packard Company*, 1996.

[20] T Jones. Linux virtualization and PCI passthrough, 2012.

[21] Jonathan Katz. *Efficient cryptographic protocols preventing "Man-in-the-Middle" attacks.* PhD thesis, Columbia University, 2002.

[22] Samuel T King and Peter M Chen. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P)*, pages 14–pp. IEEE, 2006.

[23] Kostya Kortchinsky. Cloudburst: A VMware guest to host escape story. *Black Hat USA*, 2009.

[24] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook.* Wiley Indianapolis, 2004.

[25] Vikram Kumar. Rootkit – an intruder living in your kernel. `https://www.symantec.com/connect/articles/rootkit-intruder-living-your-kernel`, August 2009. [Online; accessed 1-June-2017].

[26] Duy Le, Hai Huang, and Haining Wang. Understanding performance implications of nested file systems in a virtualized environment. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 8–8, Berkeley, CA, USA, 2012. USENIX Association.

[27] Zhenkai Liang and R Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[28] Director of Customer Reliability Engineering Luke Stone. Bringing PokemonGO to life on google cloud, 2016.

[29] Stuart McClure, Joel Scambray, and George Kurtz. *Hacking Exposed Fifth Edition: Network Security Secrets & Solutions*. McGraw-Hill/Osborne, 2005.

[30] Gary McGraw and Cigital CTO. Exploiting software: How to break code. In *Invited Talk, Usenix Security Symposium, San Diego*, 2004.

[31] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 279–294. San Diego, CA, USA, 1996.

[32] Peter Mell and Tim Grance. The NIST definition of cloud computing. 2011.

[33] Cisco Visual Networking. Cisco global cloud index: Forecast and methodology, 2012-2017,(white paper), 2013.

[34] Serkan Ozkan. CVE details: The ultimate security vulnerability datasource, 1999.

[35] Archana Pasupulati, Jason Coit, Karl Levitt, Shyhtsun Felix Wu, SH Li, JC Kuo, and Kuo-Pao Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, volume 1, pages 235–248. IEEE, 2004.

[36] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 1–20. Springer, 2008.

[37] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[38] Joanna Rutkowska. Introducing blue pill. *The official blog of the invisiblethings. org*, 22, 2006.

[39] Joanna Rutkowska. Subverting VistaTM kernel for fun and profit. *Black Hat Briefings*, 2006.

[40] Kateryna Rybina, Abhinandan Patni, and Alexander Schill. Analysing the migration time of live migration of multiple virtual machines. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science*, CLOSER 2014, pages 590–597, Portugal, 2014. SCITEPRESS - Science and Technology Publications, Lda.

[41] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM, 2007.

[42] Telecommunication Industry Association. Standards, Technology Dept, and American National Standards Institute. *Telecommunications Infrastructure Standard for Data Centers*. Telecommunication Industry Association, 2005.

[43] Virtual Open Systems. The virtual open systems video demos to virtualize ARM multicore platforms, August 2012.

[44] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract pay load execution. In *International Workshop on Recent Advances in Intrusion Detection*, pages 274–291. Springer, 2002.

[45] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 158–177. Springer, 2010.

[46] Vice President Spencer Wang. Netflix to announce second-quarter 2017 financial results, 2017.

[47] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS)*, pages 545–554. ACM, 2009.

[48] Dan Williams, Eslam Elnikety, Mohamed Eldehiry, Hani Jamjoom, Hai Huang, and Hakim Weatherspoon. Unshackle the cloud! In *The 3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2011.

[49] Rafal Wojtczuk. Adventures with a certain xen vulnerability (in the PVFB backend). *Message sent to bugtraq mailing list on October 15th*, 2008.

[50] Rafal Wojtczuk and Joanna Rutkowska. Following the white rabbit: Software attacks against Intel VT-d technology. 2011.

[51] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. Spectre: A dependable introspection framework via system management mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

Appendix A

FILEBENCH MACROS

A.1 File Server macro: fileserver.f

```
set $myset=bigfileset
set $mylog=logfiles
set $dir=/home/jconnell/testing
set $filesize=64k
set $nfiles=50000
set $meandirwidth=20
set $nthreads=50
set $nprocesses=1
set $iosize=1m
set $meanappendsize=16k

define fileset name=$myset,path=$dir,size=$filesize,entries=$nfiles,
  dirwidth=$meandirwidth,prealloc=80

define process name=filereader,instances=$nprocesses
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop createfile name=createfile1,filesetname=$myset,fd=1
    flowop writewholefile name=wrtfile1,srcfd=1,fd=1,iosize=$iosize
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=$myset,fd=1
    flowop appendfilerand name=appendfilerand1,iosize=$meanappendsize,fd=1
    flowop readwholefile name=readfile1,fd=1,iosize=$iosize
    flowop closefile name=closefile3,fd=1
    flowop deletefile name=deletefile1,filesetname=$myset
    flowop statfile name=statfile1,filesetname=$myset
  }
}
echo  "joeyconnelly File-server Version 3.0 personality loaded"
run 60
```

A.2 Mail Server macro: varmail.f

```
set $myset=bigfileset
set $mylog=logfiles
set $dir=/home/jconnell/testing
set $filesize=2k
set $nfiles=50000
set $meandirwidth=1000000
set $nthreads=16
set $nprocesses=1
set $iosize=16k
set $meanappendsize=8k

define fileset name=$myset,path=$dir,size=$filesize,entries=$nfiles,
  dirwidth=$meandirwidth,prealloc=80

define process name=filereader,instances=$nprocesses
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop deletefile name=deletefile1,filesetname=$myset
    flowop createfile name=createfile2,filesetname=$myset,fd=1
    flowop appendfilerand name=appendfilerand2,iosize=$meanappendsize,fd=1
    flowop fsync name=fsyncfile2,fd=1
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=$myset,fd=1
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize
    flowop appendfilerand name=appendfilerand3,iosize=$meanappendsize,fd=1
    flowop fsync name=fsyncfile3,fd=1
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=$myset,fd=1
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize
    flowop closefile name=closefile4,fd=1
  }
}
echo  "joeyconnelly Varmail Version 3.0 personality loaded"
run 60
```

A.3 Web Server macro: webserver.f

```
set $myset=bigfileset
set $mylog=logfiles
set $dir=/home/jconnell/testing
set $filesize=16k
set $nfiles=50000
set $meandirwidth=20
set $nthreads=100
set $nprocesses=1
set $iosize=1m
set $meanappendsize=16k

define fileset name=$myset,path=$dir,size=$filesize,entries=$nfiles,
  dirwidth=$meandirwidth,prealloc=100,readonly
define fileset name=$mylog,path=$dir,size=$filesize,entries=1,
  dirwidth=$meandirwidth,prealloc

define process name=filereader,instances=$nprocesses
{
  thread name=filereaderthread,memsize=10m,instances=$nthreads
  {
    flowop openfile name=openfile1,filesetname=$myset,fd=1
    flowop readwholefile name=readfile1,fd=1,iosize=$iosize
    flowop closefile name=closefile1,fd=1
    flowop openfile name=openfile2,filesetname=$myset,fd=1
    flowop readwholefile name=readfile2,fd=1,iosize=$iosize
    flowop closefile name=closefile2,fd=1
    flowop openfile name=openfile3,filesetname=$myset,fd=1
    flowop readwholefile name=readfile3,fd=1,iosize=$iosize
    flowop closefile name=closefile3,fd=1
    flowop openfile name=openfile4,filesetname=$myset,fd=1
    flowop readwholefile name=readfile4,fd=1,iosize=$iosize
    flowop closefile name=closefile4,fd=1
    flowop openfile name=openfile5,filesetname=$myset,fd=1
    flowop readwholefile name=readfile5,fd=1,iosize=$iosize
    flowop closefile name=closefile5,fd=1
    flowop openfile name=openfile6,filesetname=$myset,fd=1
    flowop readwholefile name=readfile6,fd=1,iosize=$iosize
    flowop closefile name=closefile6,fd=1
    flowop openfile name=openfile7,filesetname=$myset,fd=1
    flowop readwholefile name=readfile7,fd=1,iosize=$iosize
    flowop closefile name=closefile7,fd=1
    flowop openfile name=openfile8,filesetname=$myset,fd=1
    flowop readwholefile name=readfile8,fd=1,iosize=$iosize
    flowop closefile name=closefile8,fd=1
    flowop openfile name=openfile9,filesetname=$myset,fd=1
    flowop readwholefile name=readfile9,fd=1,iosize=$iosize
    flowop closefile name=closefile9,fd=1
    flowop openfile name=openfile10,filesetname=$myset,fd=1
    flowop readwholefile name=readfile10,fd=1,iosize=$iosize
    flowop closefile name=closefile10,fd=1
    flowop appendfilerand name=appendlog,filesetname=logfiles,
      iosize=$meanappendsize,fd=2
  }
}
echo  "joeyconnelly Web-server Version 3.1 personality loaded"
run 60
```

Appendix B

BASH SHELL SCRIPTS

B.1 IO Testing - Filebench

```
#!/bin/bash

#
# Adjustable script parameters.
#
TAG=_L0
FILE_NAME=FILEBENCH_raw60G_1to2G_virtio-disk_cache-none_cpu8_virtio-net_cfq
NUM_RUNS=5
SHMOO_FILE_SIZE=false
MIN=1
MAX=128
BASE=2
EXE="/usr/local/bin/filebench -f"
declare -a TEST_FILES=(
    "cloudskulk-fileserver.f"
    "cloudskulk-webserver.f"
    "cloudskulk-varmail.f"
)
TEST_DIR=filebench-tempfiles
COPY=temp
LOG=_errFile.log

#
# Non-adjustable parameters.
#
PATH_TXT='set $dir='
SIZE_TXT='set $filesize='
INPUT_TXT="bigfileset populated:"
OUTPUT_TXT="IO Summary:"
# IO testing file system on HOME path.
declare -a BENCH_FILES=(
    "$HOME/$TEST_DIR/bigfileset"
    "$HOME/$TEST_DIR/logfiles"
)
if [ $SHMOO_FILE_SIZE == false ];then
```

```
    MIN=1
    MAX=2
fi

quickPrint(){
    echo -e "$1"
    echo -e "$1" >> $2
}
removeFiles(){
    # Not removing these potentially large files from file system
    #  was found to cause performance degradation in subsequent tests.
    rm -rf ${BENCH_FILES[0]} ${BENCH_FILES[1]}
}

#
# Create initial output data files/directories.
#
rm -rf $HOME/$TEST_DIR
mkdir $HOME/$TEST_DIR
tagTime=$(date +"%m-%d")
rawFile=$FILE_NAME$tagTime$TAG.raw
csvFile=$FILE_NAME$tagTime$TAG.csv
errFile=$tagTime$LOG
startTime=$(date +"%m-%d-%Y_%H-%M-%S")
title="Run[#],TestType[test.f],Files[#],TotalSize[MB],TotalOps[#],
   Throughput[ops/s],Latency[ms/op]"
rm -f $errFile
quickPrint $startTime $csvFile
quickPrint $title $csvFile

#
# Loop on each filebench macro.
#
for currTest in "${TEST_FILES[@]}"
do
    #
    # Change home directory inside .f file.
    #
    newPathText="$PATH_TXT$HOME/$TEST_DIR"
    sed --in-place "/$PATH_TXT/c$newPathText" $currTest

    #
    # Iterate each filebench test increasing file sizes by given base.
```

```
#
for (( newSize=$MIN; newSize<=$MAX; newSize*=$BASE ))
do

    #
    # Change file size inside .f file.
    #
    if [ $SHMOO_FILE_SIZE == true ];then
        newSizeText="$SIZE_TXT$newSize"
        newSizeText+="k"
        sed --in-place "/$SIZE_TXT/c$newSizeText" $currTest
    else
        let newSize=$MAX+1
    fi

    #
    # Loop single macro, same parameters to collect average.
    #
    for (( testID=0; testID<$NUM_RUNS; testID++ ))
    do

        #
        # Execute filebench benchmark.
        #
        removeFiles
        currTime=$(date +"%H-%M-%S")
        echo -e "testing convience, time before test: $currTime"
        $EXE $currTest 1> $COPY 2>> $errFile
        currTime=$(date +"%H-%M-%S")
        echo -e "testing convience, time after test: $currTime"
        removeFiles
        ### for memory info while testing issue: df -h

        #
        # Parse output result data for key results.
        #
        numFiles=$(sed "/$INPUT_TXT/!d" $COPY | awk -F ' ' '{print $4}')
        totalSize=$(sed "/$INPUT_TXT/!d" $COPY | awk -F ' ' '{print $18}')
        totalOps=$(sed "/$OUTPUT_TXT/!d" $COPY | awk -F ' ' '{print $4}')
        throughPut=$(sed "/$OUTPUT_TXT/!d" $COPY | awk -F ' ' '{print $6}')
        latency=$(sed "/$OUTPUT_TXT/!d" $COPY | awk -F ' ' '{print $11}' |
           awk -F "ms/op" '{print $1}')
```

```
            #
            # Output results to .raw + .csv + console.
            #
            line="$testID,$currTest,$numFiles,$totalSize,$totalOps,
               $throughPut,$latency"
            quickPrint $line $csvFile
            cat $COPY >> $rawFile
        done
    done
done

#
# Clean unneeded files and save total script execution time for user convenience.
#
rm -f $COPY
endTime=$(date +"%m-%d-%Y_%H-%M-%S")
quickPrint $endTime $csvFile
```

B.2 CPU/Memory Testing - Kernel Compile

```
#!/bin/bash

#
# Adjustable script parameters.
#
TAG=_L0
FILE_NAME=KERNEL_raw60G_1to2G_virtio-disk_cache-none_cpu8_virtio-net_cfq
NUM_RUNS=3
FIRST_RUN=true
LINK="https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.0.tar.xz"
SUFFIX=".tar.xz"
TEST_DIR=$HOME
RESULTS_DIR=$(pwd)
CODE_BASE=$(echo $LINK | awk -F "/" '{print $NF}' | sed -e "s/$SUFFIX//")
DOWNLOAD=$CODE_BASE$SUFFIX
LOG=_errFile.log

quickPrint(){
    echo -e "$1"
    echo -e "$1" >> $2
}
```

```
#
# Create initial output data files.
#
tagTime=$(date +"%m-%d")
rawFile="$RESULTS_DIR/$FILE_NAME$tagTime$TAG.data"
errFile="$RESULTS_DIR/$tagTime$LOG"
startTime=$(date +"%m-%d-%Y_%H-%M-%S")
title="Run[#],startDecomp,endDecomp,totalDecomp,startComp,endComp,totalComp"
rm -f $errFile
quickPrint $startTime $rawFile
quickPrint $title $rawFile

#
# Download linux kernel in testing directory.
#
if [ ! -d $TEST_DIR ];then
    mkdir $TEST_DIR
fi
cd $TEST_DIR
if [ ! -f $DOWNLOAD ];then
    wget $LINK
fi

#
# Loop using same config file to collect average.
#
for (( i=0; i<$NUM_RUNS; i++ ))
do
    #
    # Exit early if initial data is bogus (for testing convenience).
    #
    if [ $i = 1 ];then
        echo "Do you want to continue (y|n): "
        read input_variable
        if [ "$input_variable" == "n" ];then
            rm -f $DOWNLOAD
            rm -rf $CODE_BASE
            exit 0
        fi
    fi

    #
```

```
# Make sure each run starts w/o decompressed or compiled kernel.
#
rm -rf $CODE_BASE

#
# Execute kernel decompression.
#
startDecompress=$(date +%s.%N)
tar -xvf $DOWNLOAD 2> $errFile
endDecompress=$(date +%s.%N)
totalDecompress=$(python -c "print(${endDecompress} - ${startDecompress})")
dconsole="\n\n\nDecompressRun[$i],$startDecompress,$endDecompress,
   $totalDecompress\n\n\n"
quickPrint $dconsole /dev/null

#
# Create/use common config file for all tests.
#
cd $CODE_BASE
if [ $FIRST_RUN == true ] && [ $i -eq 0 ];then
    make menuconfig
    cp .config $RESULTS_DIR
else
    cp -f $RESULTS_DIR/.config .
fi

#
# Execute kernel compile.
#
make clean
numCPUs="-j$(nproc)"
startCompile=$(date +%s.%N)
make $numCPUs 2> $errFile
endCompile=$(date +%s.%N)
totalCompile=$(python -c "print(${endCompile} - ${startCompile})")
cconsole="\n\n\nCompileRun[$i],$startCompile,$endCompile,$totalCompile\n\n\n"
quickPrint $cconsole /dev/null

#
# Print both decompression and compile results to file.
#
rtitle="Run[$i],$startDecompress,$endDecompress,$totalDecompress,
   $startCompile,$endCompile,$totalCompile"
```

```
    quickPrint $rtitle $rawFile
    cd $TEST_DIR
done

#
# Clean unneeded files and save total script execution time for user convenience.
#
rm -f $DOWNLOAD
rm -rf $CODE_BASE
endTime=$(date +"%m-%d-%Y_%H-%M-%S")
quickPrint $endTime $rawFile
```

B.3 Network Testing - Netperf

```
#!/bin/bash

#
# Adjustable script parameters.
#
TAG=_L0
FILE_NAME=NETPERF_raw60G_1to2G_virtio-disk_cache-none_cpu8_virtio-net_cfq
NUM_RUNS=1
LINK="ftp://ftp.netperf.org/netperf/netperf-2.7.0.tar.gz"
SUFFIX=".tar.gz"
TEST_DIR=$HOME
RESULTS_DIR=$(pwd)
CODE_BASE=$(echo $LINK | awk -F "/" '{print $NF}' | sed -e "s/$SUFFIX//")
DOWNLOAD=$CODE_BASE$SUFFIX
COPY=temp
LOG=_errFile.log

#
# Non-adjustable parameters.
#
OPTS="-l 60"
TEST_ADDR="-H 127.0.0.1"

quickPrint(){
    echo -e "$1"
    echo -e "$1" >> $2
}
```

```
#
# Create initial output data files.
#
tagTime=$(date +"%m-%d")
rawFile="$RESULTS_DIR/$FILE_NAME$tagTime$TAG.data"
errFile="$RESULTS_DIR/$tagTime$LOG"
startTime=$(date +"%m-%d-%Y_%H-%M-%S")
title="Run[#],RecvSockSize[B],SendSockSize[B],SendMsgSize[B],Throughput[MB/sec]"
rm -f $errFile
quickPrint $startTime $rawFile
quickPrint $title $rawFile

#
# Download netperf source code in testing directory.
#
if [ ! -d $TEST_DIR ];then
    mkdir $TEST_DIR
fi
cd $TEST_DIR
if [ ! -f $DOWNLOAD ];then
    wget $LINK
    if [ $? -ne 0 ];then
        quickPrint "Error: Download issues..." $errFile
        if [ ! -e "$RESULTS_DIR/$DOWNLOAD" ];then
            quickPrint "Error: Backup download does not exist..." $errFile
            exit -1
        fi
        cp -f $RESULTS_DIR/$DOWNLOAD $TEST_DIR
    fi

    #
    # Unzip, configure, and compile source code.
    #
    tar -xvzf $DOWNLOAD
    cd $CODE_BASE
    ./configure
    make
    sudo make install
    cd $RESULTS_DIR
fi

#
```

```
# Loop with same parameters to collect average.
#
netserver 2>/dev/null
for (( i=0; i<$NUM_RUNS; i++ ))
do
    #
    # Execute filebench benchmark.
    #
    netperf $TEST_ADDR $OPTS 1>$COPY 2>$errFile

    #
    # Parse output result data for key results.
    #
    recvSock=$(cat $COPY | tail -n 1 | awk '{print $1}')
    sendSock=$(cat $COPY | tail -n 1 | awk '{print $2}')
    sendMsg=$(cat $COPY | tail -n 1 | awk '{print $3}')
    throughput=$(cat $COPY | tail -n 1 | awk '{print $5}')

    #
    # Output results to .data + console.
    #
    results="$i,$recvSock,$sendSock,$sendMsg,$throughput"
    quickPrint $results $rawFile
done

#
# Remove unneccesary files.
#
contents=$(du $errFile | awk '{print $1}')
if [ $contents -eq 0 ];then
    quickPrint "\n\n\nNo errors occured during testing.\n\n\n" /dev/null
    rm -f $errFile
fi
rm -f $COPY
```