Pwnetizer: Improving Availability in Cloud Computing through Fast Cloning and I/O Randomization

Diego Perez-Botero

Master's Thesis

Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Master of Science in Engineering

Recommended for Acceptance
by the Department of Computer Science
Princeton University

Adviser: Ruby B. Lee

June 2013

Abstract

The rise of the Cloud Computing paradigm has led to security concerns amongst its adopters, given that resources are shared and mediated by a Hypervisor which may be targeted by rogue guest Virtual Machines (VMs) and remote attackers. We conducted a thorough analysis of the codebase of two popular open-source Hypervisors, Xen and KVM, followed by an extensive study of the vulnerability reports associated with them. Based on our findings, we propose a practical characterization of Hypervisor vulnerabilities. From this analysis, we see that more than one third of all attacks are due to I/O device emulation and that availability breaches are by far the most common security breaches, considering the cornerstone security properties of Confidentiality, Integrity and Availability.

We developed Pwnetizer, a novel VM cloning strategy, to address these weaknesses of virtualized environments. Pwnetizer facilitates increased availability by rapidly generating clone VMs that can instantly contribute to the overall throughput, as they increase the resources available to a cloud customer's applications (network bandwidth, CPU and RAM). Previously, VM Cloning research has prioritized the performance of computationally-intensive workloads by enabling the creation of transient clone VMs that depend on a master VM. Meanwhile, the few alternatives able to generate fully-independent stateful VM Clones suffer from considerable downtimes (tens of seconds), which is itself a loss of availability. A KVM-based prototype of our Pwnetizer solution is able to gracefully generate on-demand independent VM Clones with sub-second downtimes.

At minimal additional overhead, our cloning technology also randomizes the I/O device drivers employed by each clone VM. This takes advantage of the variety of device drivers with overlapping functionality supported by commodity Hypervisors. Without having to vet them beforehand, we defend a set of diversified clone VMs against current and future attacks on I/O device drivers with security vulnerabilities. This further improves availability by preventing large-scale VM crashes caused by attacks made possible by device emulation bugs.

Acknowledgements

I would like to thank my Thesis Adviser, Ruby B. Lee, for steering my research in the right direction and helping me gain a decent understanding of numerous security-related topics in such a short period of time. Our weekly meetings were an unlimited source of interesting ideas. I would also like to thank all current members of the PALMS research group (Jakub, Pramod, Fangfei and Tianwei) for the time they spent listening to my work and providing me with invaluable input. In particular, Jakub was a great mentor and collaborator, who helped fill in all the necessary voids in my unimpressive hardware knowledge.

I am infinitely grateful to Jennifer Rexford for the help and guidance that she gave me throughout this journey. I thoroughly enjoyed having her as my Academic Adviser and working alongside her as a COS 217 Teaching Assistant. I also want to express my gratitude to Dushyant Arora for his feedback along the way and for being a very supportive friend/colleague.

I would not have accomplished any of this without my family's support. My parents, Martha and Alonso, provided me with all the opportunities that any son could ever ask for, while my sister, Juliana, has always been an incredible role model and my personal idol. Last but not least, I owe my sanity to Katie, who stood by me while I faced the immense challenges that come along with studying in a world-class institution. I look forward to spending the rest of my life with her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Virtual Machines (VMs) have become commonplace in modern computing, as they enable the execution of multiple isolated Operating System instances on a single physical machine. This increases resource utilization, makes administrative tasks easier, lowers overall power consumption, and enables users to obtain computing resources on demand. Virtualized environments are at the heart of Cloud Computing and are usually implemented with the use of a Hypervisor, which is a software layer that lies between the Virtual Machines (VMs) and the physical hardware. The Hypervisor allocates resources to the VMs, such as main memory and peripherals. It is in charge of providing each VM with the illusion of being run on its own hardware, which is done by exposing a set of virtual hardware devices (e.g., CPU, Memory, NIC, Storage) whose tasks are then scheduled on the actual physical hardware. These services come at a price: Hypervisors are large pieces of software, with 100,000 lines of code or more. As a result, researchers have been tackling security concerns of traditional Hypervisors (e.g., [61]), further motivated by numerous bug reports disclosed for popular Hypervisors (e.g., Xen, KVM, OpenVZ) in a variety of software vulnerability databases, including SecurityFocus [55] and NIST's Vulnerability Database [46].

This thesis consists of three sections:

1. We conduct a Hypervisor vulnerability study in Chapter 2 that integrates three new vulnerability classifications into a set of Hypervisor-specific [trigger, attack vector, target] mappings. From the study, we find that *I/O & network device emulation* and *Availability* are the main weaknesses of commodity Hypervisors.

2. In Chapter 3, we develop a novel VM Cloning strategy for increased *availability.* Our clone VMs are fully independent, fast to create, and distributed over many Hypervisors. Better application performance is also attained as a by-product.

3. We enhance our VM Cloning mechanism with a driver randomization technique, which tackles *I/O and network device emulation vulnerabilities.* This is covered in Chapter 4.

Chapter 2

Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers

In this chapter, we characterize Hypervisor vulnerabilities through three novel classifications using Xen and KVM Code Vulnerability and Exposure (CVE) reports as our dataset. We then integrate the three classifications into a set of Hypervisor-specific [trigger, attack vector, target] mappings that indicate (1) where an attack can be initiated, (2) the Hypervisor functionality being exploited by the attack, and (3) the runtime space that is compromised after the attack's success. This will help us gain a deeper understanding of the threats that cloud customers' applications and data are exposed to, which will act as the motivation for Chapters 3 and 4.

2.1 Chapter Overview

While software vulnerability databases (e.g., SecurityFocus [55]) provide a plethora of information, not much analysis of the information has thus far been performed. Our work aims to fill this gap through an extensive study of the vulnerability reports associated with Xen and KVM.

The goal of this chapter is to characterize the security vulnerabilities of Hypervisors, based on real attacks. This chapter uses material from [49]. Our key contributions are:

1. Three classifications for Hypervisor vulnerabilities based on (1) the Hypervisor functionality where the vulnerability arises, (2) the source that triggers such vulnerability, and (3) the target that is affected by the security breach.
2. An integration of these three classifications to:
   (a) Show potential attack paths (Section 2.6).
   (b) Understand existing attacks (Section 2.7.1).
   (c) Help focus defenses (Section 2.7.2).
   (d) Assist in the discovery of new attacks (Section 2.7.3).

The rest of the chapter is organized as follows. Section 2.2 provides background on Hypervisors. Section 2.3 gives a high-level view of Hypervisor vulnerabilities. Sections 2.4, 2.5 and 2.6 describe our extensive analysis and classification of Hypervisor vulnerabilities and potential attack paths. Section 2.7 describes an existing attack, and some Hypervisor defenses that have been proposed. Section 2.8 discusses related work.

2.2 Background on Hypervisors

The virtualization marketplace is comprised of both mature (e.g., VMWare and Xen) and up-and-coming (e.g., KVM and Hyper-V) participants. Of the four main Hypervisor offerings, which take up 93% of the total market share [28], two are closed-source (VMWare and Hyper-V) and two are open-source (Xen and KVM). Recent surveys [28] [29] suggest that the number of different Hypervisor brands deployed in datacenters is broad and expanding, with a multi-Hypervisor strategy becoming the norm. As such, the percentage of datacenters actively using a specific Hypervisor to host client VMs is known as that Hypervisor's *presence*. Under that definition, VMWare has a total presence of 81%, and 52% of the datacenters use it as their primary Hypervisor, followed by Xen (81% presence, 18% as primary), KVM (58% presence, 9% as primary), and Microsoft's Hyper-V (43% presence, 9% as primary) [28] [29].

VMWare has dominated the virtualization market since its release in 2003, but some of its customers are looking elsewhere, driven primarily by cost [33]. Xen's presence and market share are expected to grow in the coming years due to the success of Amazon Web services and Rackspace, both of which use Xen as their main virtualization platform. Its market share doubled from 9% to 18% last year [33], which indicates that there is market interest in using open-source solutions. A number of vendors (including IBM, Red Hat, Intel and HP) with an interest in building an ecosystem around the other popular open-source Hypervisor, KVM, formed the Open Virtualization Alliance (OVA) [26]. To date, 241 vendors have joined the OVA [33].

We decided not to study VMWare and Hyper-V because of the dearth of public knowledge about their internals. Public Code Vulnerabilities and Exposures (CVEs) for VMWare are always from an outsider's perspective. As such, most of those CVEs focus on network attacks targeting remote management software (e.g., Cross-Site Scripting in CVE-2012-5050). Meanwhile, we were only able to find three CVEs for Hyper-V (i.e., CVE-2011-1872, CVE-2010-3960 and CVE-2010-0026), which does not constitute a representative sample set. Consequently, we have decided to focus on Xen and KVM. Considering Xen's and KVM's influence over the virtualization marketplace, with 81% and 51% datacenter presence respectively, understanding their vulnerabilities can benefit millions of users worldwide.

Below, we briefly summarize Xen's and KVM's architectural traits and their different Hypervisor designs.

2.2.1 Xen

Xen is a very well-known Open Source Hypervisor, in use since 2003. As shown in Figure 2.1, Xen is a Type-I (bare metal) Hypervisor, running directly on top of the hardware and managing all of the host's resources. It also has a privileged VM named Dom0, which carries out all of the VM management actions (e.g., start, stop and migrate guest VMs). The Dom0 VM is a full custom-tailored Linux kernel that is aware of the Xen deployment, whereas the normal guest VMs usually run in full virtualization mode (HVM mode), which emulates the entire system (i.e., BIOS, HDD, CPU, NIC) and does not require any modifications to the guest OS. In addition to basic administrative tasks, Dom0 exposes the emulated devices by connecting an instance of a device emulator (i.e., QEMU) to each guest VM.

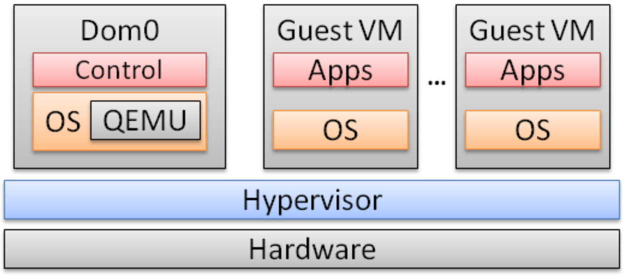


Figure 2.1: Xen Architecture.

2.2.2 KVM

KVM is a relatively new open-source project, which dates back to Red Hat's acquisition of Qumranet in 2008. Its adoption has spiked since it was made part of the main Linux kernel branch starting from version 2.6.20, becoming the main virtualization package in Ubuntu, Fedora, and other mainstream Linux operating systems. From Figure 2.2, one can identify many differences with Xen. Each guest VM runs as a separate user process and has a corresponding QEMU device emulation instance running with it. The Hypervisor itself runs as a module inside a host Operating System, which makes KVM a Type-II (hosted) Hypervisor.

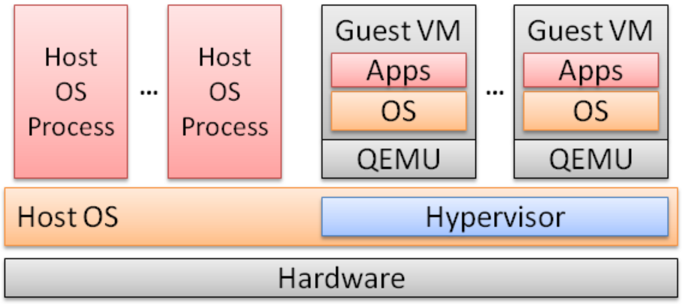


Figure 2.2: KVM Architecture.

2.2.3 QEMU

QEMU is a mature device emulator. Both KVM and Xen use QEMU's code to present virtual devices to their guest VMs, including but not limited to optical drives, graphics cards, and network cards. This is a key piece of software that provides the generic peripherals that an OS expects (e.g., mouse, keyboard, hard disk) and carries out the translations between those devices and the physical ones.

2.2.4 Hardware Virtualization Features

Intel supports the virtualization of x86 processor hardware by way of an additional set of architectural features (i.e., new instructions and control data structures) referred to as Virtual-Machine Extensions (VMX) [30]. While the Hypervisor runs in VMX root mode, guest VMs run in VMX non-root mode. This allows the Hypervisor to retain control of processor resources, given that the non-root operations are restricted. Throughout a VM's lifetime, certain sensitive instructions (e.g., CPUID, GETSEC, INVD) and events (e.g., exceptions and interrupts) cause *VM Exits* to the Hypervisor. These VM Exits are handled by the Hypervisor, who decides the appropriate action to take and then transfers control back to the VM via a *VM Entry.* A Virtual Machine Control Data Structure (VMCS) stores the data needed by the Hypervisor to restore the guest VM's state once it has handled the VM Exit and also contains information regarding the VM Exit's cause and nature. Most key concepts are mirrored almost identically in AMD's x86 virtualization mechanism (AMD-V).

2.3 Overview of Vulnerabilities

We searched a set of well-known vulnerability databases for reports regarding KVM and Xen: NIST's National Vulnerability Database (NVD) [46], SecurityFocus [55], Red Hat's Bugzilla [52] and CVE Details [10]. Fortunately, all vulnerability reports are assigned a unique CVE Identifier by a CVE Numbering Authority (CNA) and all CNAs use the MITRE Corporation as an intermediary to guarantee the uniqueness of their identifiers, making it easy to eliminate duplicate reports. According to the CVE reports, 59 vulnerabilities have been identified in Xen and 38 in KVM as of July 15, 2012.

Successful exploitation of a vulnerability leads to an attack, which can hinder the Confidentiality, Integrity, or Availability of the Hypervisor or one of its guest VMs. Each CVE report explicitly indicates the type of security breach that it can lead to as a combination of those three security properties. Roughly 50% of vulnerabilities reported so far can lead to security breaches in all three fronts. The second most common effect of exploiting these vulnerabilities is to only pose a threat to the *availability* of the Hypervisors (Denial of Service). This makes sense, taking into account that a bug in a Hypervisor module will most likely lead to an unforseen state, which often manifests itself in the form of a guest VM crash or, in the worst case, a complete host crash.

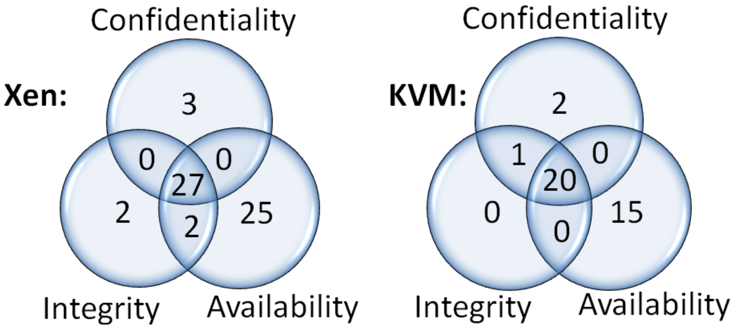


Figure 2.3: Vulnerability type breakdown for Xen (left) and KVM (right).

2.4 Hypervisor Functionalities as Attack Vectors

To better understand the different vulnerabilities, we identified the 11 functionalities that a traditional Hypervisor provides and mapped vulnerabilities to them:

1. Virtual CPUs
2. Symmetric Multiprocessing (SMP)
3. Soft Memory Management Unit (MMU)
4. Interrupt and Timer Mechanisms
5. I/O and Networking
6. Paravirtualized I/O
7. VM Exits
8. Hypercalls
9. VM Management (configure, start, pause and stop VMs)
10. Remote Management Software
11. Hypervisor Add-ons

Categories 1 through 6 present the virtualized hardware infrastructure that VMs require to operate properly. VM Exits and Hypercalls (Categories 7 and 8) are mechanisms through which VMs can delegate sensitive operations to the Hypervisor. Category 9 deals with facilities needed by the Hypervisor to manage VM state. Category 10 deals with non-essential remote management, while Category 11 allows optional add-on modules to the Hypervisor. We further explain these categories below. The CVE reports mentioned throughout this section are listed in Appendix A.

① **Virtual CPUs**: A set of virtual CPUs (vCPUs) is assigned to each guest VM being hosted by a Hypervisor. The state of each of these vCPUs is saved to and loaded from their respective VM's Virtual Machine Control Structure (VMCS) guest-state area. Since vCPUs must mirror a physical CPU's actions for each and every machine language instruction, the Hypervisor must handle register states appropriately and schedule vCPU tasks to the physical CPUs while making any necessary translations back and forth.

CVE-2010-4525 is an example of a disclosure of Hypervisor memory contents through vCPU registers because of an incomplete initialization of the vCPU data structures, where one of the padding fields was not zeroed-out. Given that the memory for the data structure is allocated in kernel space, the padding field might end up containing information from data structures previously used by the Hypervisor.

② **Symmetric Multiprocessing (SMP)**: Hypervisors can host guest VMs with SMP capabilities, which leads to the possibility of two or more vCPUs belonging to a single VM being scheduled to the physical CPU cores in parallel. This mode of operation adds complexity to the management of guest VM state and requires additional precautions at the moment of deciding a vCPU's Current Privilege Level (CPL – e.g., Ring 0 or Ring 3).

SMP vulnerabilities arise from Hypervisor code making assumptions that only hold true on single-threaded processes. For example, CVE-2010-0419 refers to a bug that permitted malicious Ring 3 processes to execute privileged instructions when SMP was enabled because of the presence of a race condition scenario. To do so, they would invoke a legitimate I/O instruction on one thread and attempt to replace it with a privileged one from another thread right after KVM had checked its validity, but before it was executed.

③ **Soft MMU**: Guest VMs cannot be granted direct access to the MMU, as that would allow them to access memory belonging to the Hypervisor and other co-hosted VMs. Under the absence of a virtualization-aware hardware MMU, such as Extended Page Tables (EPT), a Soft MMU is run by the Hypervisor to maintain a shadow page table for each guest VM. Every page mapping modification invoked by a VM is intercepted by the Soft MMU so as to adjust the shadow page tables accordingly.

Vulnerabilities in the Soft MMU's implementation are dangerous because they may lead to the disclosure of data in arbitrary address spaces, such as a co-hosted guest VM's memory segment or the Hypervisor's memory segment. In the specific case of CVE-2010-0298, KVM's emulator always uses Ring 0 privilege level when accessing a guest VM's memory on behalf of the guest VM's code. Given that MMIO instructions are emulated, an unprivileged (Ring 3) application running inside a VM can leverage access to an MMIO region (e.g., framebuffer) to trick KVM into executing a malicious instruction that modifies that same VM's kernel-space (Ring 0) memory.

④ **Interrupt and Timer Mechanisms**: A Hypervisor must emulate the interrupt and timer mechanisms that the motherboard provides to a physical machine. These include the Programmable Interval Timer (PIT), the Advanced Programmable Interrupt Controller (APIC), and the Interrupt Request (IRQ) mechanisms.

In the case of CVE-2010-0309, lack of validation of the data contained in the PIT-related data structures enabled a rogue VM to cause a full host OS crash, a serious denial-of-service attack.

⑤ **I/O and Networking**: The Hypervisor also emulates I/O and networking. Xen and KVM make device emulation possible through division of labor, by having two types of device drivers. Front-end drivers reside inside the guest VMs and run in Ring 0, providing the usual abstraction that the guest OS expects. Nonetheless, those drivers cannot access physical hardware directly, given that the Hypervisor must mediate user accesses to shared resources. Therefore, front-end drivers communicate with back-end drivers, which have full access to the underlying hardware, in order to fulfill the requested operations. In turn, back-end drivers enforce access policies and multiplex the actual devices. KVM and Xen employ QEMU's back-end drivers by default.

I/O & network device emulation is usually implemented in higher-level languages (e.g., C and C++), so the data abstractions are richer but more dangerous when hijacked. Very elaborate attacks are enabled by the expressiveness of higher-level languages like C. For example, CVE-2011-1751 describes a bug that was used to develop the Virtunoid attack [17]. QEMU tried to hot-unplug whichever device the programmers desired, regardless of the device's support for hot-unplugging. Therefore, the lack of state cleanup by some virtual devices resulted in *use-after-free* opportunities, where data structures that were previously being used by a hot-unplugged virtual device remained in memory and could be hijacked with executable code by an attacker.

⑥ **Paravirtualized I/O**: paravirtualized VMs run modified guest kernels that are virtualization-aware and use special hypercall APIs to interact with the Hypervisor directly. Paravirtualization of I/O operations decreases the number of transitions between the guest VM and the Hypervisor, resulting in performance gains. This scenario requires special front-end and back-end drivers which are not necessarily developed by the same vendor as the one responsible for regular device emulation (e.g., QEMU).

Paravirtualized I/O vulnerabilities and emulated I/O vulnerabilities are very much alike. They are rooted in the interactions between front-end and back-end drivers, as well as those between back-end drivers and the outside world. For instance, CVE-2008-1943 describes a vulnerability in Xen that allowed paravirtualized front-end drivers to cause denial-of-service conditions and possibly execute arbitrary code with Dom0 privileges. This could be done by sending a malicious shared framebuffer descriptor to trick Xen into allocating an arbitrarily large internal buffer inside Dom0.

⑦ **VM Exits** are the mechanism used by the Hypervisor to intercept and carry out operations invoked by guest VMs that require Virtual Machine eXtensions (VMX) root privileges. These VM-to-Hypervisor interfaces are architecture-dependent (e.g., different code for x86 than for AMD64) and are very well specified in the architecture manuals. They are usually implemented using low-level programming languages (Assembly or Machine language), relying on restrictive bitwise operations. For Intel VT-x, this code is the one supporting all operations described in chapters 23 through 33 of Intel's Software Developer's Manual [30].

The fact that VM Exit-handling code does not possess very rich data structures means that vulnerabilities hardly have any exploitable effects other than a host or guest VM crash (Denial-of-Service). For example, all VMCS fields have a unique 32-bit field-encoding, which rules out common vulnerabilities that arise from variable-size input, such as buffer overflows. According to CVE-2010-2938, requesting a full VMCS dump of a guest VM would cause the entire host to crash when running Xen on a CPU without Extended Page Table (EPT) functionality. The reason for this was that Xen would try to access EPT-related VMCS fields without first verifying hardware support for those fields, allowing privileged (Ring 0) guest VM applications to trigger a full denial-of-service attack on certain hosts at any time.

⑧ **Hypercalls** are analogous to system calls in the OS world. While VM Exits are architecture-specific (e.g., AMD64, x86), hypercalls are Hypervisor-specific (e.g., Xen, KVM) and provide a procedural interface through which guest VMs can request privileged actions from the Hypervisor. For example, hypercalls can be used to query CPU activity, manage Hard Disk partitions, and create virtual interrupts.

Hypercall vulnerabilities can present an attacker, who controls a guest VM, with a way to attain escalated privileges over the host system's resources. Case in point, CVE-2009-3290 mentions the fact that KVM used to allow unprivileged (Ring 3) guest callers to issue MMU hypercalls. Since the MMU command structures must be passed as an argument to those hypercalls by their physical address, they only make sense when issued by a Ring 0 process. Having no access to the physical address space, the Ring 3 callers could still pass random addresses as arguments to the MMU hypercalls, which would either crash the guest VM or, in the worst case, read or write to kernel-space memory segments.

⑨ **VM Management** functionalities make up the set of basic administrative operations that a Hypervisor must support. The configuration of guest VMs is expressed in terms of their assigned virtual devices, dedicated PCI devices, main memory quotas, virtual CPU topologies and priorities, etc. The Hypervisor must then be able to start, pause and stop VMs that are true to the configurations declared by the cloud provider. These tasks are initiated by Xen's Dom0 and KVM's Libvirt toolkit [38].

Kernel images must be decompressed into memory and interpreted by the management domain when booting up a VM. CVE-2007-4993 indicates that Xen's bootloader for paravirtualized images used Python exec() statements to process the custom kernel's user-defined configuration file, leading to the possibility of executing arbitrary python code inside Dom0. By changing the configuration file to include the line shown in Listing 2.1, a malicious user could trick Dom0 into issuing a command that would trigger the destruction of another co-hosted domain (substituting *id* with the victim domain's ID).

```
default ”+str(os.system(”xm destroy id”))+”
```

Listing 2.1: Contents of /boot/grub/grub.conf for an attack on Dom0 with a user-provided kernel

⑩ **Remote Management Software**: These pieces of software are usually web applications running as a background process and are not essential for the correct execution of the virtualized environment. Their purpose is generally to facilitate the Hypervisor's administration through user-friendly web interfaces and network-facing virtual consoles.

Vulnerabilities in these bundled applications can be exploited from anywhere and can lead to full control over the virtualized environment. For example, CVE-2008-3253 describes a Cross-Site Scripting attack on a remote administration console that exposed all of Xen's VM management actions to a remote attacker after stealing an administrator's authentication cookies.

⑪ **Hypervisor Add-ons**: Hypervisors like Xen and KVM have modular designs that enable extensions to their basic functionalities – Hypervisor Add-ons. For example, the National Security Agency (NSA) has developed their own version of Xen's Security Modules (XSM) called FLASK.

Hypervisor add-ons increase the likelihood of Hypervisor vulnerabilities being present, since they increase the size of the Hypervisor's codebase. For example, CVE-2008-3687 describes a heap overflow opportunity in one of Xen's optional security modules, FLASK, which results in an escape from an unprivileged domain directly to the Hypervisor.

2.4.1 Breakdown of Vulnerabilities

We analyzed all of KVM's and Xen's CVE reports from the 4 vulnerability databases, labeling each with its functionality-based attack vector. Our resulting vulnerability breakdowns are presented in Table 2.1. It can be observed that the I/O Device Emulation categories (i.e., I/O and Networking along with Paravirtualized I/O) account for more than one third of the known vulnerabilities for each of the Hypervisors (33.9% of Xen's and 39.5% of KVM's vulnerabilities). This can be attributed to the variety of back-end drivers that are supported by both Xen and KVM. A normal QEMU installation is capable of emulating all sorts of virtual devices (e.g., NIC, display, audio) and different models of each type (Intel Ethernet i82559C, Realtek Ethernet rtl8139, etc.), leading to a considerable number of distinct use cases and a fairly large codebase.

Table 2.1: Breakdown of known vulnerabilities by Hypervisor functionality for Xen and KVM.

Attack Vector	**Xen**	**KVM**
Virtual CPUs	5 (8.5%)	8 (21.1%)
SMP	1 (1.7%)	3 (7.9%)
Soft MMU	4 (6.8%)	2 (5.3%)
Interrupt and Timer Mechanisms	2 (3.4%)	4 (10.5%)
I/O and Networking	11 (18.6%)	10 (26.3%)
Paravirtualized I/O	9 (15.3%)	5 (13.2%)
VM Exits	4 (6.8%)	2 (5.3%)
Hypercalls	2 (3.4%)	1 (2.6%)
VM Management	7 (11.9%)	2 (5.3%)
Remote Management Software	9 (15.3%)	1 (2.6%)
Hypervisor add-ons	5 (8.5%)	0 (0.0%)
Total	**59**	**38**

The number of Remote Management Software vulnerabilities in Xen (accounting for 15.3% of its vulnerabilities) shows that non-essential services may increase the attack surface significantly. More interestingly, KVM reports a markedly lower contribution from VM Management vulnerabilities towards the total (5.3% in KVM vs 11.9% in Xen). This might suggest that KVM's architectural decision of running the libvirt toolkit (in charge of VM Management functionalities) as an additional module inside Hypervisor space is more secure than Xen's decision of allocating an entire privileged VM (Dom0) for the same purpose. After all, Xen's Dom0 domain is a specialized linux kernel, which means that it needs to execute at least a minimal set of OS services in order to run, therefore increasing the likelihood of bugs.

2.5 Further Characterization of Hypervisor Vulnerabilities

Our analysis of KVM and Xen vulnerability reports gave rise to two additional complementary classifications: trigger source and attack target. A Hypervisor vulnerability manifests itself inside a Hypervisor module's code, but can be triggered from a variety of runtime spaces and can target one or more of those runtime spaces. Listed from lowest to highest privilege level: (1) Network, (2) Guest VM's User-Space, (3) Guest VM's Kernel-Space, (4) Dom0/Host OS, (5) Hypervisor.

The trigger source and attack target are of great importance when assessing a vulnerability's ease of exploitability and impact, respectively. The trigger source can be determined by comparing the restrictions of each of the runtime spaces with the execution rights required to reproduce the vulnerability. Since these five categories correspond to hierarchical privilege levels, we show the least possible privilege level for the trigger source, and the greatest possible privilege level for the attack target in Tables 2.2 and 2.3.

2.5.1 Trigger Sources and Attack Targets

① **Network**: This is the least privileged runtime space, but also the easiest to attain. Any remote user can initiate an attack on a Hypervisor and its guest VMs if it is located in a subnet from which the machine running the Hypervisor is reachable.

② **Guest VM's User-Space**: Almost any code can be executed from a guest VM's Ring 3; however, some functionalities are reserved for the OS or the Hypervisor (causing an exception). Nevertheless, it is easiest to get user-space code to run, so any exploits from this ring are attractive to an attacker. For example, CVE-2010-4525 mentions an attack from a guest VM's Ring 3 involving the CPUID x86 instruction.

③ **Guest VM's Kernel-Space**: Injecting malicious OS-level (Ring 0) code requires compromising the OS security. Interestingly, in IaaS cloud deployments, tenants can simply lease VMs and run their OS of choice – one which may already be malicious. For example, CVE-2008-1943 mentions an attack from a Guest VM's Kernel-Space, as it requires control over the paravirtualized front-end driver.

④ **Dom0/Host OS**: Some runtime spaces have privilege levels that lie between those of a guest VM's OS and the ones possessed by the Hypervisor. In Xen's case, Dom0 is a privileged VM with direct access to I/O and networking devices. At the same time, Dom0 is allowed to invoke VM Management operations. While KVM does not have a Dom0 equivalent, the fact that the Hypervisor is part of a fully-operational Linux kernel gives way to other types of threats (e.g., local users in the host system).

⑤ **Hypervisor**: This is the most desired runtime space because it has Ring -1 privileges, so any command can be run from this space. The Hypervisor can access any resource in the host system (i.e., memory, peripherals, CPU state, etc), which means that it can access every guest VM's resources.

2.5.2 Breakdown of Vulnerabilities

As can be observed in Table 2.2, the most common trigger source is the Guest VM User-Space (Ring 3), accounting for 39.0% of Xen's and 34.2% of KVM's vulnerabilities. This is worrying, as it indicates that any unprivileged guest VM user has the necessary privileges to pose a threat to the underlying Hypervisor. The Guest VM Kernel-Space is the second most common trigger source, with roughly 32% of the total in both cases. Hence, 71.2% of all Xen and 65.8% of all KVM vulnerabilities are triggered from a guest VM. Also note that there are no vulnerabilities with Hypervisor space as their trigger source, which makes sense because an attacker who has control over the Hypervisor already has the maximum privilege level attainable.

Table 2.2: Breakdown of known vulnerabilities under trigger source classification for Xen and KVM.

Trigger Source	**Xen**	**KVM**
Network	11 (18.6%)	2 (5.3%)
Guest VM User-Space	23 (39.0%)	13 (34.2%)
Guest VM Kernel-Space	19 (32.2%)	12 (31.6%)
Dom0/Host OS	6 (10.2%)	11 (28.9%)
Hypervisor	0 (0.0%)	0 (0.0%)
Total	**59**	**38**

Two differences between the two Hypervisors stand out: Xen is much more vulnerable to network-based attacks than KVM, but KVM is more sensitive to Host OS-based attacks. The first observation follows from our attack vector analysis (Section 2.4), which showed that Remote Management Software vulnerabilities are a big problem for Xen. On the other hand, KVM's sensitivity to Host OS threats is to be expected because, being part of the main Linux kernel branch, its code can be invoked by other kernel-space processes running on the host, leaving it exposed to malicious privileged local users.

Table 2.3: Breakdown of known vulnerabilities under target-based classification for Xen and KVM.

Attack Target	**Xen**	**KVM**
Network	0 (0.0%)	0 (0.0%)
Guest VM User-Space	0 (0.0%)	0 (0.0%)
Guest VM Kernel-Space	12 (20.3%)	9 (23.7%)
Dom0/Host OS	25 (42.4%)	11 (28.9%)
Hypervisor	22 (37.3%)	18 (47.4%)
Total	**59**	**38**

It can be observed from Table 2.3 that Dom0 is a more common target than the Hypervisor in Xen, whereas KVM shows the opposite behaviour (its Host OS is less common than the Hypervisor as a target). This difference between the two Hypervisors is due to the location of the I/O Device Emulation back-end drivers, which are found in Dom0 with Xen and in the Hypervisor with KVM. The I/O & Network Device Emulation functionalities contribute more than one third of the known vulnerabilities in both Hypervisors, so the location of the back-end drivers has great influence over the relative distribution of vulnerabilities among the possible attack targets.

2.6 Hypervisor Attack Paths

Tables 2.4 and 2.5 show an integration of all of our three attack classifications for Xen and KVM, respectively. Each row illustrates a potential attack path; starting at some trigger source, exploiting a specific Hypervisor functionality, to attack a set of targets. In each row, the trigger sources are less privileged software entities, while the attack targets are the more privileged software entities, thus enabling privilege escalation. A co-located hostile VM can take multiple iterations through the attack paths (left to right, wraparound to left to right, etc.) to achieve privilege escalation, eventually attaining Dom0/Host OS or Hypervisor-level privileges. When the attacker achieves these elevated privileges, it can see, modify or deny services to a victim VM, thus breaching the victim's confidentiality, integrity or availability.

The specially marked Ⓧs in Table 2.4 are an example of a possible 2-step privilege escalation path that a privileged guest VM user (Ring 0) could follow in order to reach Hypervisor runtime space (Ring -1) in a Xen deployment. The first step would be to exploit a *VM Management* vulnerability to gain control of Dom0. A viable attack to achieve this transition is CVE-2007-4993 (see Section 2.4), which enables the execution of arbitrary python code inside Dom0. Once in control of Dom0, exploiting a *Soft MMU* vulnerability could grant the malicious user control over the most desirable runtime space: Ring -1. The Q35 attack (CVE-2008-7096), covered in the next section, could be used to that end.

Table 2.4: Xen's Vulnerability Map in tabular form

Trigger Source				**Attack Vector**	**Attack Target**		
NW	Usr	OS	Dom0		OS	Dom0	HV
	X		X	Virtual CPUs	X		X
	X			SMP	X		
	X	X	Ⓧ	Soft MMU	X		Ⓧ
	X	X		I&T. Mech.			X
X	X	X		I/O and NW	X	X	
	X	X		Paravirt. I/O	X	X	
	X	X		VM Exits	X		X
		X		Hypercalls			X
	X	Ⓧ	X	VM Management		Ⓧ	X
X				Rem. Mgmt. SW			X
X		X	X	HV add-ons		X	X

Table 2.5: KVM's Vulnerability Map in tabular form

Trigger Source				**Attack Vector**	**Attack Target**		
NW	Usr	OS	Host		OS	Host	HV
	X	X	X	Virtual CPUs	X	X	X
	X			SMP	X	X	
	X	X		Soft MMU	X		X
	X	X	X	I&T. Mech.		X	X
X	X	X	X	I/O and NW	X	X	X
X		X		Paravirt. I/O	X	X	X
	X			VM Exits	X	X	
	X			Hypercalls	X		
			X	VM Management			X
X				Rem. Mgmt. SW			X
				HV add-ons			

Legend: NW = Network; Usr = Guest VM User-Space; OS = Guest VM Kernel-Space; Host = Host OS; HV = Hypervisor; I&T Mech. = Interrupt and Timer Mechanisms; Paravirt. I/O = Paravirtualized I/O; Rem. Mgmt. SW = Remote Management Software

2.7 Case Studies and Defenses

The goal of our exploration of the vulnerabilities, their classification, and creation of the vulnerability maps, is to help researchers better understand attacks on Hypervisors and determine where defenses should be concentrated. We now first show how two real world attacks can be analyzed using the maps.

2.7.1 Understanding Existing Attacks

For our case studies, we present one Xen and one KVM attack. The first is the Dom0 Attack on Xen from Black Hat USA 2008 [54]. The second is the Virtunoid Attack on KVM from DEFCON/Black Hat 2012 [17].

Case Study: Dom0 Attack on Xen (Black Hat USA 2008)

This attack [54] revolves around Intel's Q35 chipset for the Core 2 Duo/Quad platforms. In Q35 chipsets, the processor provides the capability to re-claim the physical memory overlapped by the Memory Mapped I/O logical address space. Under normal operation, the REMAPBASE and REMAPLIMIT registers are calculated and loaded by the BIOS. The amount of memory remapped is the range between Top of Low Usable DRAM (TOLUD) and 4 GB. This physical memory will be mapped to the logical address range defined between the REMAPBASE and the REMAPLIMIT registers. The end result is shown in Figure 2.4.

The Invisible Things team [54] managed to hijack Xen's Hypervisor memory from the Dom0 domain in Q35 chipsets by exploiting the host system's remapping registers. Xen's Dom0 has write access to the host system's REMAPBASE and REMAPLIMIT registers, so a malicious Dom0 kernel is able to set up a memory remapping range pointing to the Hypervisor's physical memory, as shown in Figure 2.5. As a result, the Dom0 attacker can read and modify the Hypervisor's memory space during runtime. This constitutes a serious confidentiality and integrity breach, making it possible for the attacker to run any series of instructions with Ring -1 privilege level.

Note that the Invisible Things team [54] employed an already-compromised Dom0 kernel to carry out the Q35 attack. Therefore, in terms of the Xen vulnerability map shown in Table 2.4, the initial trigger source for the original Q35 attack is Dom0 (which runs in Ring 0). The attack vector is via part of the Soft MMU because memory management and memory control is mediated incorrectly. A mechanism to lock the remapping registers after the BIOS has set them up is a possible defense, eliminating the possibility for a rogue Dom0 domain to write new values to them. Finally, the attack target is the Hypervisor (Ring -1), whose memory space is completely compromised.

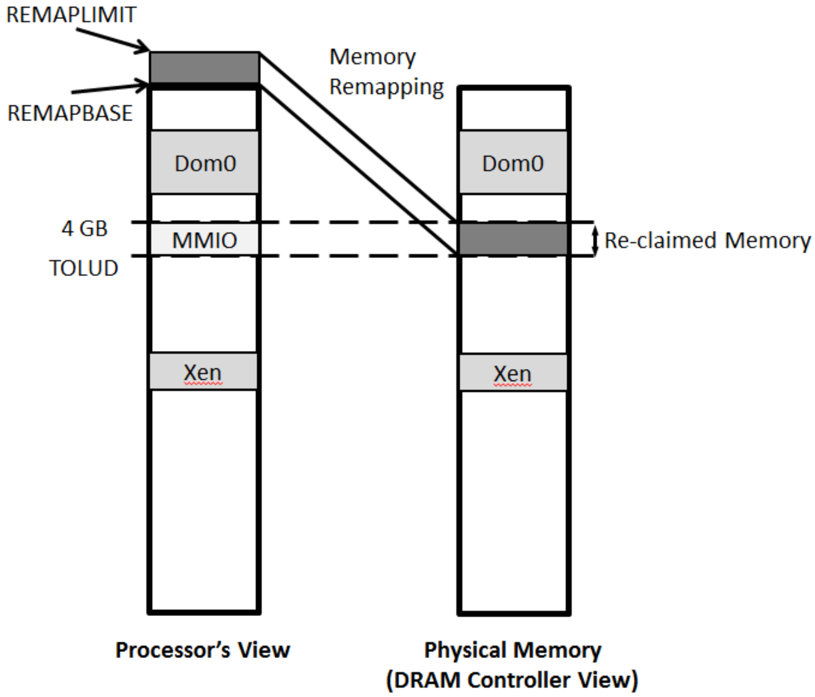


Figure 2.4: Memory remapping during normal operation of Q35 chipset.

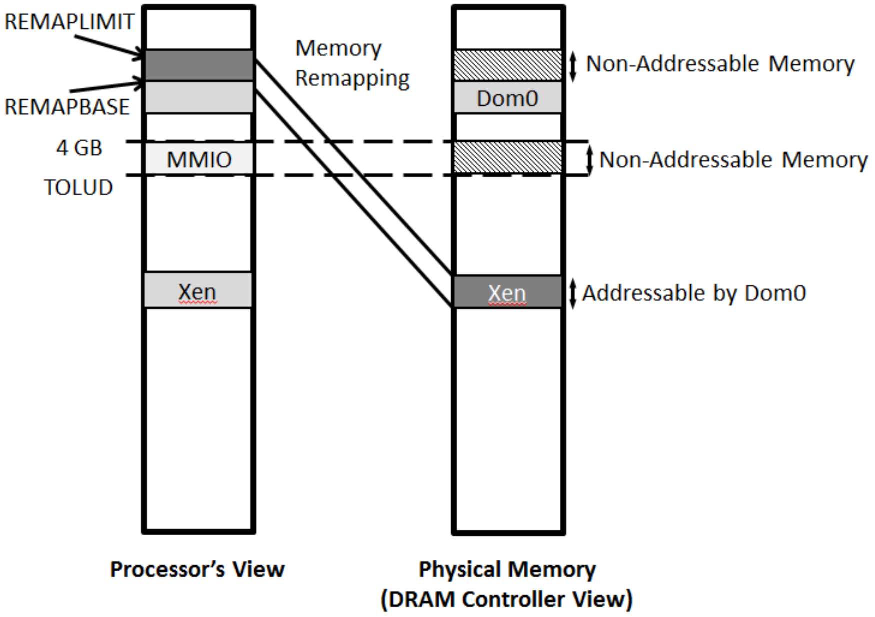


Figure 2.5: Memory remapping during attack on Q35 chipset.

Case Study: Virtunoid Attack (DEFCON/Black Hat 2011)

The Virtunoid attack [17] illustrates a combination of two attack steps that achieve privilege escalation to Hypervisor privilege level in KVM. The first step exploits an I/O hot-unplug vulnerability, and the second step uses this to achieve privilege escalation. The specific vulnerability exploited was CVE-2011-1751.

PIIX4 is the default South Bridge emulated by KVM and supports PCI hot-plugging and hot-unplugging. A security researcher discovered that KVM's emulated RTC (real time clock, a type of clock source for the operating system) was not designed to be hot-unplugged. A use-after-free opportunity occurs when the virtual RTC is hot-unplugged, as it leaves around dangling pointers to `QEMUTimer` objects. Dangling pointers store references to objects that are no longer valid. This happens when objects are freed from memory and pointers referencing them are not updated to reflect those changes. `QEMUTimer` objects are part of a circular linked list that is continuously traversed by following the `next` struct pointer (see Listing 2.2). The `opaque` struct field points to the function that is to be invoked once the timer expires. Therefore, if an attacker managed to inject data into the memory segment previously occupied by a `QEMUTimer` object belonging to the virtual RTC, the object's opaque field could be overwritten to point to malicious code, which would in turn be automatically executed by the PIIX4 emulator after some time.

```
struct QEMUTimer {
  QEMUClock *clock;
  int64\_t expire\_time;
  QEMUTimerCB *cb;
  void *opaque;
  struct QEMUTimer *next; };
```

Listing 2.2: QEMUTimer struct definition in KVM.

One way to inject data into the memory space previously occupied by a `QEMUTimer` object is to take advantage of the fact that QEMU emulates a variety of other devices. Therefore, any emulated device requiring new memory in the stack immediately after the hot-unplugging of the virtual RTC will most likely be allocated the same memory segment that has just been freed (where the `QEMUTimer` object was located at). The Virtunoid attack takes advantage of the emulated Network Card, given that it responds to ICMP Echo (Ping Request) packets by synchronously generating a second packet whose contents can be controlled through the contents of the incoming packet (RFC 1122 states that data received in the Echo Request must be entirely included in the Echo Reply). Consequently, the emulated Network Card provides a way for the attacker to trigger calls to `malloc` by the QEMU process, as the second packet (Echo Reply) requires memory in the stack. The basic steps followed for the memory hijacking to take place are summarized in Figure 2.6.

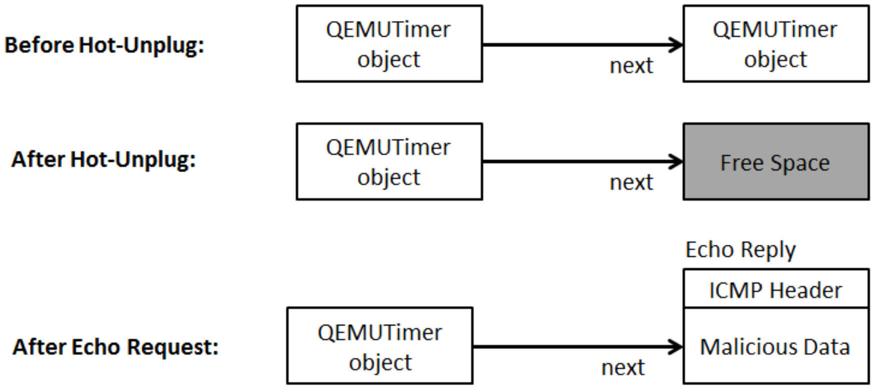


Figure 2.6: Series of events that leads to virtual RTC's `QEMUTimer` object being hijacked in Virtunoid attack.

Once the `QEMUTimer` object is hijacked, the opaque pointer can be changed to reference any code found in the QEMU memory space, so hard-coded QEMU functions and even code sent in other ICMP packets could be invoked through that pointer.

Referring to the KVM vulnerability map in Table 2.5, the initial trigger source for the Virtunoid attack is a Guest VM's Kernel-Space. In order to request the hot-unplugging of QEMU's RTC timer, the attacker must write the value *2* to I/O port `0xae08`, which requires root (Ring 0) privileges inside a guest VM. The attack vector is *I/O and Networking*, since the presence of dangling `QEMUTimer` pointers is attributed to errors in the code related to I/O device emulation. Finally, the attack target is the Hypervisor, given that KVM runs its back-end drivers (main QEMU process) inside the Hypervisor runtime space. After this first attack path through Table 2.5, no privilege escalation has occurred, as the attacker remains in Guest VM Kernel-Space.

Privilege Escalation occurs during the second step of the attack (a second pass through Table 2.5). The trigger source for this step can be the Guest VM User-Space or even the Network, since ICMP Echo Request packets do not require any special privileges besides having internet connectivity. The attack vector is again the *I/O and Network device emulation* functionality, as it requests stack memory in a predictable and controllable way. The attack target is the Hypervisor. When this step is completed, the memory segment that has been hijacked belongs to the Hypervisor (Ring -1) and allows the attacker to run arbitrary code with Hypervisor privilege level, so a privilege escalation condition has been attained. Therefore, the attacker ends up transitioning from Guest VM Kernel-Space to Hypervisor runtime space. From then onwards, any Hypervisor functionality or runtime space can be reached by the attacker through clever manipulation of the opaque pointer.

It is worth noting that the actual privilege escalation is from the Guest VM kernel code on the first step to the Hypervisor code at the end of the second step.

2.7.2 Helping Focus Defenses

While we believe this to be the first categorization of Hypervisor vulnerabilities, researchers have been aware of various attacks on Hypervisors and have proposed a number of defenses. We summarize some of these, then suggest areas to focus defenses based on our Hypervisor vulnerability maps.

One defense strategy against attacks is to make the Hypervisor codebase more resilient to attacks. Projects such as HyperSafe [67] have looked at hardening the code to make it more difficult to inject code and subvert the control flow of the Hypervisor through clever programming techniques. This aims to address attack paths targeting the Hypervisor; however, this does not mean that all attack vectors with the Hypervisor as the attack target are mitigated.

Protecting the Hypervisor kernel from an untrusted management OS [37] is another approach that has been proposed. Such work covers Dom0/Host OS trigger sources, and especially paths with a VM Management attack vector.

Another defense strategy is to use hardware-assisted techniques for protecting the software integrity of the Hypervisor to detect the attacks before they can do damage. For example, Copilot [50] employs a special purpose PCI device to read the physical memory of the target system. HyperCheck [66] looks at using features of the microprocessor, namely the system management mode (SMM) to inspect the Hypervisor. HyperSentry [3] also used the SMM to bypass the Hypervisor for integrity measurement purposes. Such work aims to cover paths toward the Hypervisor attack target.

A fourth defense strategy is removing the Hypervisor altogether. The NoHype [31, 61] architecture for Cloud Computing eliminates the Hypervisor layer and places VMs directly on top of the physical hardware while still being able to start, stop and run multiple VMs at the same time.

From our analysis of CVEs, we believe that defenses for commodity Hypervisors should start by focusing on Hypervisor correctness. Thorough input validation, proper tracking of context changes, complete initialization of control structures, complete clearing of sensitive data on process termination, and full awareness of the underlying hardware's capabilities would immediately reduce the Hypervisor's attack surface. The emulation of I/O and networking devices proves to be a common point of failure, so Hypervisor vendors should aim at developing a small set of secure back-end drivers instead of trying to provide a large number of virtual devices with overlapping functionality (e.g., e1000, ne2k_pci and rtl8139 networking cards) that are hard to maintain.

2.7.3 Assisting in the Discovery of New Attacks

While many proposed defenses exist, numerous paths through the Hypervisor vulnerability maps (Tables 2.4 and 2.5) are not yet covered. However, it should be noted that some of them can be dismissed. For example, it is inconceivable for a remote (network-bound) attacker to directly exploit VM Exit-related vulnerabilities because VM Exits are a mechanism that only exists in the boundary between a VM and the Hypervisor, so the attacker must first gain access to a VM. Even though some attack paths can be ruled out, most of them are valid. The absence of current attacks with a specific [source, vector, target] combination does not necessarily rule out the possibility of a future attack leading to those conditions. For instance, KVM's vulnerability map (Table 2.5) does not report any existing attacks with a VM's Kernel-Space as the source and the Hypervisor as a target using Hypercalls as an attack vector (i.e., [OS, Hypercalls, HV]), which we know for a fact to be a possibility judging from Xen's vulnerability map (Table 2.4).

Our vulnerability maps provide a way to assess the coverage of each protection mechanism that a cloud provider can employ. Equally important, they provide a way of identifying weak spots before an actual attack surfaces. If a given [source, vector, target] combination is not addressed by a secure Hypervisor, a malicious user will be able to decide where to concentrate his efforts. Conversely, our work suggests specific areas where the cloud provider can focus hardening efforts to minimize the risk of such attacks.

2.8 Related Work

To the best of our knowledge, there has been no detailed categorization of Hypervisor vulnerabilities as presented in this chapter. Many researchers have looked at security issues in Cloud Computing and produced surveys of those issues. The surveys (e.g., [62], [71]) focus on various threats for the cloud environment as a whole: abuse of Cloud Computing resources, insecure APIs, etc. There has also been work on classification of threats based on the different service delivery models of Cloud Computing [57]. Other works have presented classifications of security issues at different levels, such as network, host or application [4].

As one of its contributions, our work aims to categorize different attack vectors. Outside of Cloud Computing, researchers have explored categorizing kernel-level rootkits to aid future detection [36]. Others have looked at attack surfaces in Cloud Computing, however, at the level of user, services and cloud without diving into details of the attack surfaces on the virtualization layer itself [23]. Attack surface inflation [21] has been explored, including the change of the attack surface as new components are integrated into an existing system (e.g., adding virtualization). Researchers have also looked at the classification of threats and challenges faced by different components of an IaaS (infrastructure-as-a-service) Cloud Computing deployment – components such as cloud software, platform virtualization, network connectivity, etc. [11]. Different from all these works, our work focuses on the Hypervisor attack surface.

Interesting work on mapping cloud infrastructure [53] has given insights on how to find a specific target cloud server to attack. There have not been, however, other works which aim, as we do, to map the cloud infrastructure attack paths.

Chapter 3

Availability Fueled by Instantaneous VM Cloning

In Chapter 2, we identified **Availability** and **I/O and network device emulation** as the two main security weaknesses of commodity Hypervisors (see Sections 2.3 and 2.4.1). In this chapter, we try to deal with *availability* issues caused by the use of commodity Hypervisors in a Cloud Computing context through the development of a new VM Cloning strategy.

3.1 Motivation

When addressing availability concerns, we find that VM Cloning is a viable strategy. We know for a fact that Hypervisors can be compromised in many ways, so we do not want a client's applications and data to depend on a single Hypervisor instance. VM Cloning is interesting because it allows us to have a VM's state present in many different hosts at once. It is also faster than powering on a new VM, since you avoid having to load the operating system and all of the system services from scratch, which requires a considerable amount of disk reads and CPU time. Furthermore, VM Cloning replicates the runtime state, including all changes made to a VM's configuration since the moment it was turned on. Thus, a clone VM is fully-functional without requiring any intervention on behalf of the cloud customer. Replicated runtime state also means that software caches are preserved, so the original VM's pre-cloning performance levels can almost instantly be reached by the clone VM.

3.1.1 VM Cloning for Availability

There are two availability goals that we want to achieve:

- **Resilience to crashes**: service disruption and data loss should be minimized when a VM or Hypervisor instance goes down.
- **Micro-elasticity**: if a VM does not possess enough resources (i.e., RAM, disk, CPU, network bandwidth) to provide acceptable performance to its users, it should be possible to dynamically increase its available resources in a timely manner and with minimal service disruption.

Resilience to crashes is the most obvious property of VM Cloning. If a VM and its clones are guaranteed to be consistent with each other, one of the clones can quickly take over the VM's responsibilities without any loss of data if a crash occurs. To fulfill this availability requirement, passive (suspended) clones are ideal, as they can be kept fully-consistent by just transferring the original VM's main memory pages over the network. Keeping the clones powered down helps us avoid networking conflicts and makes it so that a single disk file (for secondary storage) is sufficient because only one live VM will be reading and writing to disk at any given time. Considering that a clone is only needed to be turned on after the original VM is compromised, the passive cloning scheme is simple and effective when it comes to *crash resilience.*

Micro-elasticity is a more ambitious availability goal. Once a VM is instantiated, the RAM and CPU resources allocated to it cannot be changed without restarting the VM[1]. Given that the detection of RAM and CPU resources happens during boot time, the VM's OS does not contemplate the possibility of those hardware characteristics changing thereafter. This means that, if a VM is running a workload that (1) requires more RAM than what is available to it (leading to page swapping and degraded performance) or (2) runs unbearably slowly under 100% CPU utilization, the VM's users are left with two options: (1) power off the VM, increase its RAM and/or CPU allocation, then restart, resulting in considerable downtime or (2) keep using the VM despite its unsatisfying performance, giving way to sub-optimal quality of service. A VM's network bandwidth suffers from other limitations – the host's network card imposes a physical limit to the amount of network bandwidth that the VM can utilize. If the host's network card reaches its saturation point,

[1] Memory Ballooning [65] is not a RAM *hot add* feature; it is limited by a VM's original RAM allocation. The balloon is just a process that runs inside the VM and expands/contracts its memory usage, leaving less/more space for the other applications running inside the VM. The balloon's expansion can be used to force the VM to swap out some memory pages to disk and reduce its main memory footprint, which is useful when oversubscribing RAM on top of a host. However, a 2GB VM with memory ballooning will never have more than 2GB of RAM available to it during its lifetime. If it needs more than 2GB of total RAM, one must power it off, increase its RAM allocation, and boot it up again.

the VM's networked applications will encounter long delays and/or packet loss, which inevitably leads to sub-optimal user experience. Live VM migration is a way of dealing with such a scenario, but the VM will always end up being limited by a single host's network bandwidth.

With VM Cloning, one can envision the possibility of generating clone VMs that are actually active and, thus, contribute to the overall throughput of the original VM's internal workloads, providing us with *micro-elasticity*. Considering that each clone VM is given the same amount of CPU, RAM and network resources as the original VM, this would allow us to effectively multiply the resources available to a cloud customer's applications under the presence of appropriate collaboration mechanisms (e.g., network load balancers and application-layer task distribution frameworks). However, we would like to preserve the *crash resilience* of passive cloning. Due to the fact that each active clone VM will have to be writing to a separate disk file and attain a unique network identity for them not to conflict with each other, every clone VM's security-critical data will end up diverging from the original VM's security-critical data unless they have some way of synchronizing the data that should not be lost after a VM's crash. This, which seems to be a disadvantage of active cloning, can actually be seen as an advantage – instead of replicating the original VM's entire state, active clone VMs are given the option to decide what state they care about and eliminate unnecessary network traffic generated by the replication of non-critical state (e.g., memory pages used by stateless services). Section 3.3 includes a more detailed treatment of data replication, concurrency, and consistency issues that arise with active cloning.

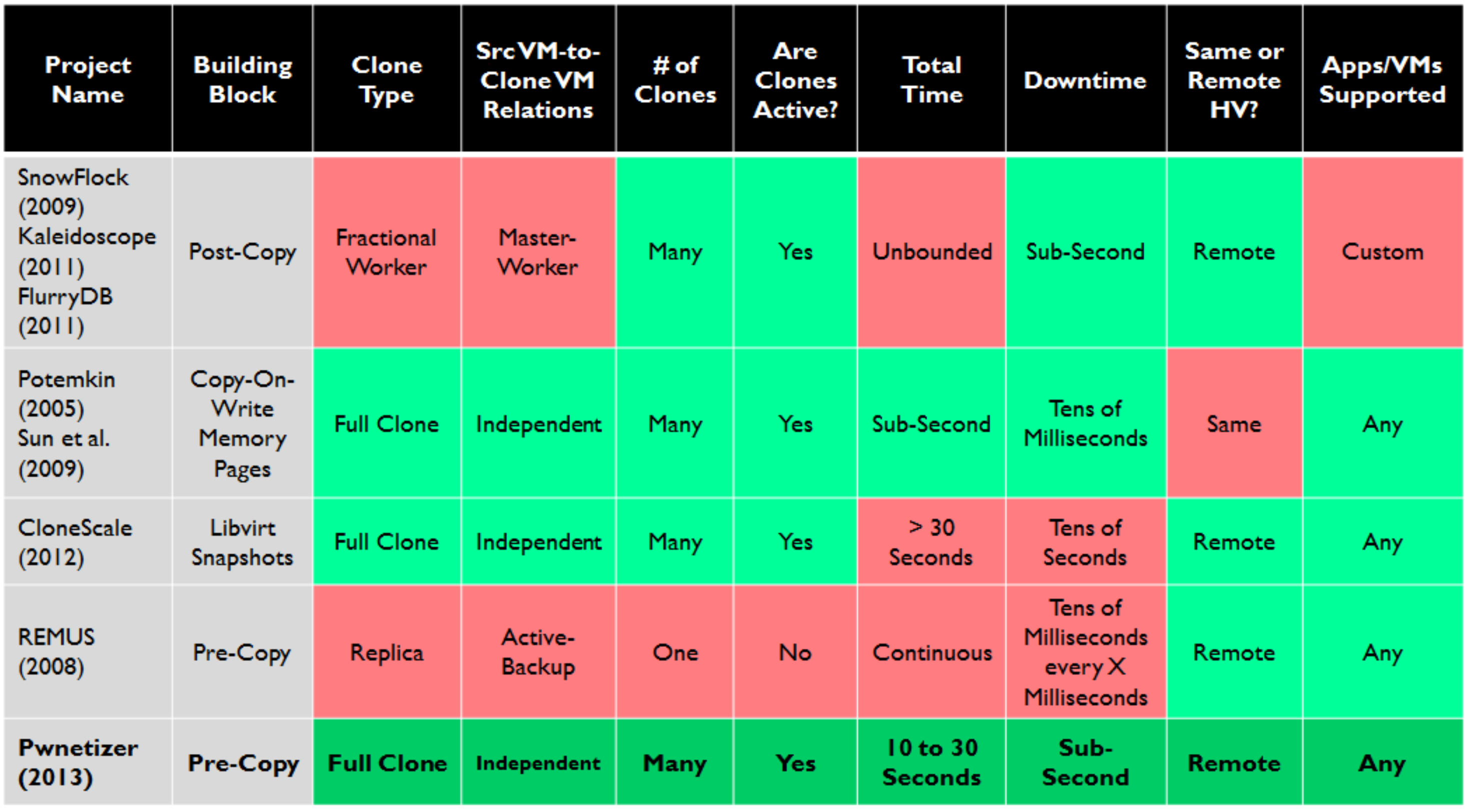
From this discussion, we can conclude that active VM clones can help us fulfill both of our availability objectives (*crash resilience* and *micro-elasticity*) when proper inter-clone collaboration and data replication mechanisms are in place. We will focus on developing an efficient VM Cloning strategy for the creation of active VM clones; researchers in the area of distributed systems have already come up with solutions for collaboration and data replication. Our cloning technique should provide us with the following consistency and independence guarantees between the original VM and its clones:

- Main memory consistency at cloning time, independence after cloning.
- Persistent storage consistency at cloning time, independence after cloning.
- No networking conflicts.

The rest of this chapter describes how we successfully developed Pwnetizer, a cloning approach that quickly generates VM clones while satisfying those guarantees.

3.2 Background on VM Cloning

If we look at existing VM Cloning work, we can identify four main trends, whose characteristics are summarized in Figure 3.1. Cells highlighted in red are weaknesses from a security perspective, while those highlighted in green indicate advantages. For a detailed description of the listed works, please refer to Section 3.9. To accomplish our availability goals, what we are looking for is a mechanism with all the properties listed on the last row. Ideally, clones should be fully independent, fast to create, and distributed over many Hypervisors.



Project Name	Building Block	Clone Type	Src VM-to-Clone VM Relations	# of Clones	Are Clones Active?	Total Time	Downtime	Same or Remote HV?	Apps/VMs Supported
SnowFlock (2009) Kaleidoscope (2011) FlurryDB (2011)	Post-Copy	Fractional Worker	Master-Worker	Many	Yes	Unbounded	Sub-Second	Remote	Custom
Potemkin (2005) Sun et al. (2009)	Copy-On-Write Memory Pages	Full Clone	Independent	Many	Yes	Sub-Second	Tens of Milliseconds	Same	Any
CloneScale (2012)	Libvirt Snapshots	Full Clone	Independent	Many	Yes	> 30 Seconds	Tens of Seconds	Remote	Any
REMUS (2008)	Pre-Copy	Replica	Active-Backup	One	No	Continuous	Tens of Milliseconds every X Milliseconds	Remote	Any
Pwnetizer (2013)	**Pre-Copy**	**Full Clone**	**Independent**	**Many**	**Yes**	**10 to 30 Seconds**	**Sub-Second**	**Remote**	**Any**

Figure 3.1: Summary of previous cloning work.

The first group of works (SnowFlock [34], Kaleidoscope [5] and FlurryDB [42]) focuses on computationally-intensive workloads. They create small single-purpose VMs that depend on a master VM's main memory pages and die off after they complete a specific task. While the clones are generated with little downtime, the cloning procedure is never-ending because of the ongoing master-worker relation. Given that we want to avoid a single point of failure, this sort of strategy is not very effective. Furthermore, the application code needs to be modified to use a specific cloning-related API, which limits the deployability of these mechanisms.

All other works aim at producing complete VM clones as opposed to single-purpose ones. The second row in Figure 3.1 lists projects that aim at generating clone VMs as fast as possible on top of the same host – Potemkin [64] and Sun *et al.* [59]. This is accomplished by powering up new VMs comprised of copy-on-write pages. While this is great for creating large numbers of VMs with minimal memory footprint, all of the clones rely on a single Hypervisor instance. Hence, these cloning strategies do not satisfy our *crash resilience* requirements.

A third approach, materialized in CloneScale [58], ends up generating fully-independent clones on remote Hypervisors. Sadly, its performance figures are lacking, with downtimes in the order of tens of seconds. In this case, the cloning procedure itself becomes an availability threat.

The fourth and most common type of VM Cloning has to do with keeping passive backups of a VM in case it crashes. Once the VM crashes, the replica goes up and replaces it. This is what Cully *et al.*'s REMUS [9] and most commercial products currently offer (e.g., Hyper-V Replica, VMWare vSphere Replication), but the performance impact can be high depending on the sampling rate. For example, using REMUS to create replication checkpoints with a frequency of 40 times per second translates into a 103% performance overhead for computational workloads [9]. In addition, clone VMs are always suspended, so we lose the *micro-elasticity* property of cloning. Lastly, REMUS can only keep one replica per active VM, which limits the levels of redundancy that we can reach. Two consecutive crashes within a short time window could still lead to the loss of security-critical data.

3.3 Pwnetizer - Orthogonal Issues

Some issues may arise when creating clones of VMs that are running certain types of applications. If state changes (e.g., writes to disk) made by one VM are expected to cascade to the other clone VMs, the cloud customer must employ frameworks with appropriate concurrency and consistency guarantees for his applications. For instance, two clones running a normal MySQL database will end up giving inconsistent sets of results after different database updates are executed on each of them. If this is not the expected behavior, a solution would be to use MySQL Cluster [45] – a cluster-friendly version of MySQL that fixes those symptoms. Many papers have tackled these sort of issues and we do not aim at solving age-old questions related to distributed systems.

Valid approaches to handling concurrency in distributed applications include, but are not limited to, Tame's [32], SEDA's [68], and Flash's [48] event-driven architectures. On the distributed state management front, examples of different levels of filesystem redundancy and system-wide consistency (weak vs strong) are found in Harp's [39] replicated filesystem, Lamport's [35] event-ordering algorithm, Chubby's [6] distributed filesystem locks, COPS's [40] casual consistency in key-value stores, and Dynamo's [14] loosely-consistent key-value stores.

Another orthogonal problem is that having a single Network File System (NFS) share to store the disk files used by all VMs in a Cloud does not scale well. However, the main insights presented in Section 3.5.3 still hold true for deployments using scalable networked shares (i.e., clustered file systems), such as Oracle Cluster File System (OCFS), VMWare's Virtual Machine File System (VMFS) and Cluster Shared Volumes (CSV).

The simplest types of workloads that handle cloning gracefully are those that are read-only. Thus, Big Data processing frameworks (e.g., Cajo [7], MapReduce [13], Piccolo [51], CIEL [44], Spark [70]) work out-of-the-box. Other safe workloads are those that are self-contained or stateless, such that local changes made by a single VM do not need to be reflected by the other VMs. P2P network nodes (e.g., Skype, BitTorrent, Gnutella), firewalls (e.g., Snort), and static websites (e.g., Apache HTTP Server) are examples of those.

3.4 Pwnetizer - Initial Considerations

3.4.1 Live Migration as a Starting Point

At first glance, VM Cloning sounds relatively simple. *Why not just conduct Live VM Migration and leave the source VM turned on at the end?* This would leave us with two VMs with the same main memory contents. However, both VMs would be writing to the same disk file, which is problematic because normal Operating Systems assume that they have dedicated access to the local hard disks. As a result, neither of the VMs would be able to determine what the other VM may have written to disk, which inevitably leads to data loss and corruption after some time. For two VMs to be able to properly share the same disk file, a clustered filesystem and a custom OS would be required. Nonetheless, such arrangement would mean that a single compromised VM could take over the secondary storage used by all other clone VMs, which is undesirable.

Suppose that we decide to leave the source VM turned on after migration. For us to rule out persistent storage conflicts and guarantee VM independence, we require a *disk cloning* mechanism to ensure that the original VM and the clone VM are writing to separate disk files, which must be completely identical at cloning time and can diverge thereafter. Unfortunately, given that disk files represent a VM's entire filesystem (i.e., OS files, application binaries and data, and user documents), they can be several Gigabytes in size. For the cloned disk file to be fully-consistent with respect to the clone VM's main memory state, the *disk cloning* procedure must be performed during live migration's *stop-and-copy* phase, when both VMs are suspended and no writes are being issued to the original VM's disk. Consequently, disk cloning becomes the main downtime bottleneck, taking into account that modern hard disks have write speeds in the order of 50 MB/s, which means that making a copy of a 1GB disk file would require roughly 20 seconds of downtime. Our Pwnetizer cloning strategy minimizes this downtime by maintaining a local mirror of the networked filesystem share, which will be covered in Section 3.5.3.

Leaving the source VM turned on after Live VM Migration leads to another problem: the two VMs will end up having the exact same network configuration (i.e., MAC and IP addresses), which translates into connectivity issues for both. Hence, one of them must acquire a new network identity to avoid networking conflicts. The main challenge in this case is for the clone VM to detect that cloning has happened so that it can begin reconfiguring its network. A way of doing this is described in Section 3.5.5.

From this discussion, it is clear that Live VM Migration serves as a good starting point for VM Cloning, as it results in a new VM with fully-consistent main memory state. Nonetheless, two non-trivial challenges (*secondary storage independence* and *network reconfiguration*) must be dealt with for the two VMs (original and clone) to coexist with each other.

3.4.2 Precopy vs Postcopy

In this section, we summarize the two most popular memory migration algorithms and evaluate their suitability for the VM Cloning scenario.

Pre-Copy

The *pre-copy* algorithm proposed by Clark *et al.* [8] keeps the source VM running for most of the migration procedure. It uses an iterative push phase, followed by a minimal stop-and-copy phase. The iterative nature of the algorithm is the result of what is known as *dirty pages*: memory pages that have been modified in the source VM since the last page transfer must be sent again to the destination VM. At first, iteration i will be dealing with less dirty pages than iteration $i-1$. Unfortunately, the available bandwidth and workload characteristics will make it so that some pages will be updated at a faster rate than the rate at which they can be transferred to the destination VM. At that point, the stop-and-copy procedure must be executed. The stop-and-copy phase is when the CPU state and any remaining inconsistent pages are sent to the new VM, leading to a fully consistent state.

Post-Copy

Post-copy migration defers the memory transfer phase until *after* the VM's CPU state has already been transferred to the target and resumed there. As opposed to *pre-copy*, where the source VM is powered on during the migration process, *post-copy* delegates execution to the destination VM. In the most basic form, *post-copy* first suspends the migrating VM at the source node, copies minimal processor state to the target node, resumes the virtual machine at the target node, and begins fetching memory pages from the source over the network. Variants of post-copy arise in terms of the way pages are fetched. The main benefit of the post-copy approach is that each memory page is transferred *at most once*, thus avoiding the duplicate transmission overhead of pre-copy [25].

Suitability for VM Cloning

The key difference between VM Migration and VM Cloning is that the former switches computation from one host to the other, whereas the latter must end up with computation running on both hosts. Figure 3.2 contrasts the pre-copy and post-copy procedures' timelines. From a performance standpoint, post-copy hinders the VM's performance the most because page faults must be fetched over the network, which takes something in the order of milliseconds as opposed to the nanoseconds that local DRAM accesses take. On the other hand, the total amount of data transferred is larger in the case of pre-copy due to its iterative nature; post-copy transfers each memory page only once because page dirtying is taking place at the destination side. Consequently, it is hard to decide between pre-copy and post-copy solely based on performance.

VM Cloning puts two new dimensions on the table: secondary storage consistency and networking conflicts. At the end of VM Cloning, both VMs must have internet connectivity and be assigned a secondary storage device that coincides with their operating system's view of the filesystem. The easiest way to tackle connectivity in VM Cloning is to leave the original VM's network configuration untouched and have the clone VM drop all ongoing sessions and assume a different IP+MAC address set. Under the pre-copy scheme, the source VM remains active while its memory pages are transferred, so connectivity is not an issue. Meanwhile, the post-copy scheme initially pauses the source VM and starts the clone VM, requiring an immediate update in the LAN's packet forwarding rules (through a gratuitous ARP reply) to preserve connectivity and keep TCP sessions up and running on the clone VM. At the end, the source VM will have to assume a new network identity, given

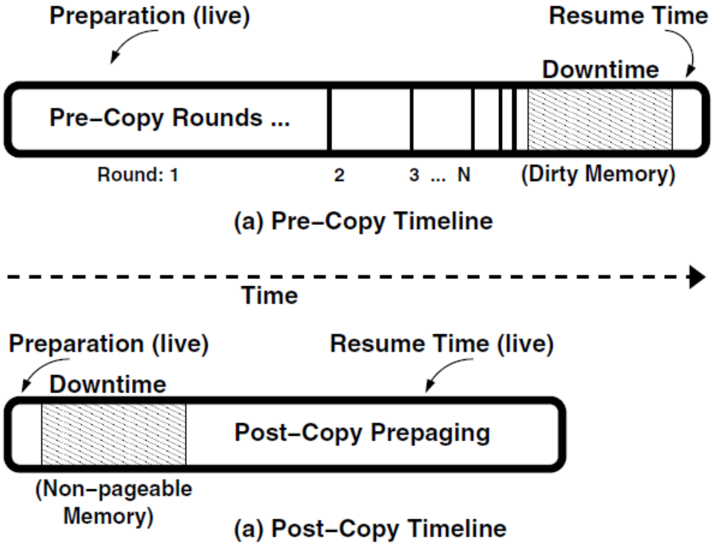


Figure 3.2: The timeline of (a)*pre-copy* vs (b)*post-copy* migration. Taken from [25].

that its clone will have taken its place inside the network. Of course, this is burdensome because packet forwarding rules have to be forcefully updated to redirect TCP/IP packets to the clone VM even though the source VM is to come alive once again. Nevertheless, this inconvenience might be justified by the guarantee that post-copy provides in terms of network efficiency (i.e., each memory page needs to be transferred only once).

Secondary storage consistency is the defining criterion when evaluating suitability in the VM Cloning scenario. When employing pre-copy, page dirtying is happening on the source VM's side; hence, by the time the clone VM comes alive, both VMs will possess a main memory state that is consistent with the source VM's secondary storage. Thus, the problem in that case lies in making a copy of that secondary storage for the clone VM to use. Post-copy further complicates things by having the dirtied pages in the clone VM's side once the process is over, which means that the clone VM's memory state will be consistent with a secondary storage that does not correspond to the one that existed when the source VM was paused. Consequently, the source VM will *travel back in time* when it is resumed and will require a copy of the secondary storage made at the beginning of the cloning process. This is clearly suboptimal, as it will force some computations that have already been carried out by the clone VM to be executed once more inside the source VM. It may also trigger system clock issues inside the source VM as a result of having paused its Operating System for an extended period of time (several seconds), which is a problem when maintaining application logs. Simply updating the clock time is not a viable solution, as it will lead to time-based jobs (e.g., cron jobs on UNIX) being skipped. In addition, mechanisms to speed up the clock to get to the correct time (e.g., ntpdate on UNIX) require a reliable NTP server or Hypervisor support (e.g., VMware Tools) and are OS-dependent.

Taking all factors into account, the pre-copy page transfer algorithm seems to fit better with the VM Cloning scenario than its counterpart. For this reason, Pwnetizer will extend pre-copy rather than post-copy in order to materialize full VM Cloning with negligible downtime.

3.5 Pwnetizer - Implementation Details

In this section, we will discuss the logic behind key design decisions that had to be made when implementing our first Pwnetizer prototype. The main objectives for our prototype were high deployability and minimized downtime. As such, already-existing technologies should be leveraged as much as possible and the time period during which the source VM must be paused should be minimal.

3.5.1 Code Structure

Most commodity Hypervisors use the pre-copy algorithm to carry out live migration of VMs, including VMware, Xen, KVM, VirtualBox, Microsoft Hyper-V and OpenVZ. Given that, as mentioned in Section 3.4.2, the aforementioned algorithm is well-suited for full VM Cloning, the most straightforward approach to materializing Pwnetizer is to extend a Hypervisor's live migration module. Ideally, our code should be cross-compatible amongst a wide range of Hypervisors, which is why we opted to modify Libvirt instead of a specific Hypervisor.

Libvirt is an open-source management tool for virtualized platforms. It runs in kernel space (Ring 0) and presents a Hypervisor-agnostic API to perform common tasks (e.g., stop, resume and launch VMs) on any supported Hypervisor. Internally, each Hypervisor-specific code module is known as a *driver*. The drivers available at the moment cover 8 popular virtualization platforms, including Xen, KVM, VMware, VirtualBox and Hyper-V. All drivers use Libvirt's Hypervisor-independent core logic. This enables Libvirt users to declare guest domains (i.e., VMs), storage pools, storage volumes, and host devices in a generic XML format. During runtime, the XML files and Libvirt's management console commands are interpreted and translated appropriately to the underlying Hypervisor by its corresponding Libvirt driver.

The Pwnetizer Libvirt extension is divided into two separate modules: a Hypervisor-independent module and a modified KVM driver, where the former comprises most of the codebase and can be reused by any future Pwnetizer-enabled driver implementations. The choice of KVM as the base Hypervisor for our prototype comes from the fact that KVM's adoption has spiked since it was made part of the main Linux kernel branch starting from version 2.6.20, becoming the main virtualization package in Ubuntu, Fedora, and other mainstream Linux distributions. Figure 3.3 shows the architecture of a KVM host system running Libvirt. The management user interface (virsh) sends Libvirt commands to the Libvirt daemon, which in turn communicates with KVM to perform the corresponding actions. The runtime behaviour is similar with other virtualization configurations and Pwnetizer support for other Hypervisors can be easily implemented because of its modular design.

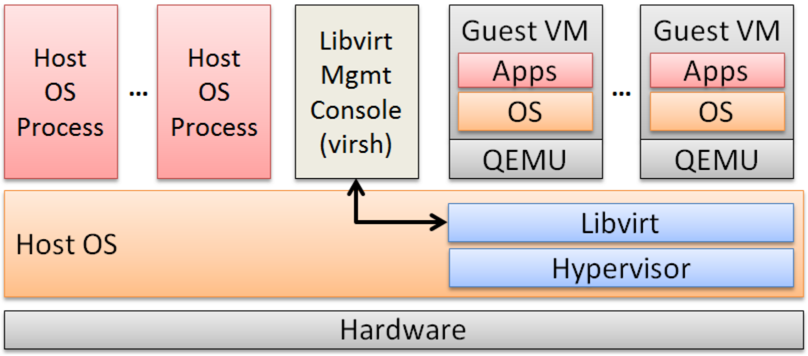


Figure 3.3: KVM+Libvirt Architecture.

3.5.2 Main Memory

VM's Memory Space

Section 3.4.2 explains why the pre-copy algorithm is the most appropriate page transfer mechanism when conducting on-demand full VM Cloning with negligible downtime. Thus, all of the issues related to main memory consistency between a VM and its clone are solved by reusing existing live migration code.

Hypervisor's Memory Space

Hypervisors store state associated with every guest VM that they are hosting. For example, KVM loads a full description of each running VM in main memory, with details regarding CPU and RAM allocation, the architecture being emulated (e.g., x86, AMD64), and the VM's peripherals (e.g., network devices, storage devices, video cards, optical drives). This description also includes domain metadata, such as the domain's name and its unique identifier (or UUID).

The main purpose of Pwnetizer's VM Cloning mechanism is to obtain an independent clone of a specific VM. This does not necessarily mean that both VMs will end up having the same exact domain description. As will be covered in Section 3.5.3, the clone VM's filesystem must be redirected to a disk file other than the one originally used by the source VM. Other changes that must be made include modifying the MAC addresses of network cards (refer to Section 3.5.4) and giving the clone VM a new domain name and UUID. Therefore, the Hypervisor responsible for hosting the clone VM will have to alter the runtime domain object that resides in the Hypervisor's memory space if it is to provide an appropriate virtualized environment for that VM. Hence, even though main memory consistency is the expected outcome at the VM level when conducting full VM Cloning, some controlled inconsistencies will have to arise at the Hypervisor level to ensure the independence of the resulting clone VM with respect to its source VM.

3.5.3 Secondary Storage

VM's Persistent Storage

Secondary storage is not a concern when carrying out VM migration, given that a single VM will be reading and writing to disk at any given time during and after the migration process. This does not hold true in the case of full VM Cloning, where two different VMs will result from the procedure and both will want to read and write to their secondary storage. If secondary storage of one of the VMs is not redirected to a new (yet consistent) disk file, the original filesystem will become corrupted shortly after in the sense that neither of the VMs will have a correct view of the filesystem's state. Our main challenge in this case is to achieve full consistency between each VM's Operating System and its persistent storage without hindering the liveliness of our cloning process. In other words, we need to preserve the negligible downtime offered by Live VM Migration while still duplicating the relevant networked file shares in a consistent manner.

Figures 3.4 through 3.8 summarize our proposed disk cloning technique, which complies with the aforementioned consistency and liveliness constraints. We use a Linux package called rsync to maintain a mirror of the NFS share being used by the VMs (see Figure 3.4). Rsync uses delta encoding to synchronize two files that contain a similar, but not identical, version of the same data. Consequently, the utility needs to transfer relatively little data to synchronize the files if it is invoked every time a new write is issued on the NFS share. For simplicity, we run an rsync thread that invokes the mirroring command every second. The key insight is that, if both the NFS share and its local mirror are kept in different directories inside the same filesystem, a disk file can be moved from the local mirror into the NFS share almost instantly, as such operation only requires modifications to the filesystem's metadata (i.e., unix inode tables); actual file data does not have to be physically copied to the other folder. Consequently, our mirroring arrangement allows us to provide the illusion of creating a copy of an arbitrary-sized disk file in sub-second time, as will be explained below.

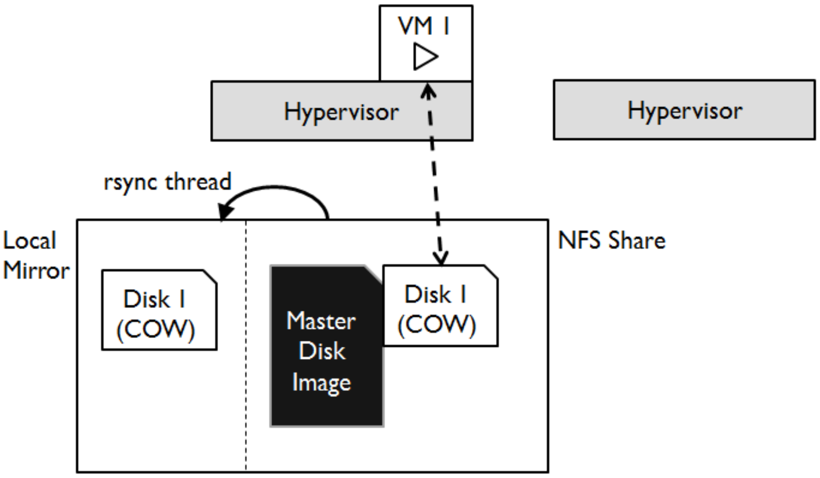


Figure 3.4: Storage state before cloning operation.

For storage efficiency, our virtual machines use QCOW2 (QEMU copy-on-write) disk images derived from a master disk image. The master disk image contains a basic OS installation, so it provides the standard file structure (approximately 4.5 GB of data for an Ubuntu Desktop installation) that all VMs initially require. The copy-on-write images grow and diverge from each other as data is written by the VMs. Considerable storage savings are achieved with this arrangement as the number of VM clones increases.

The first step in order to clone a VM is to allocate a new instance of the same VM on a remote Hypervisor and keep it paused. We call this second instance the *clone VM*, while the original VM is referred to as the *source VM*. As Figure 3.5 illustrates, the clone VM's secondary storage must be redirected to a disk file other than the one being used by the source VM, but with the same contents. This preparation step can be executed without requiring any data to be copied by simply invoking a move+rename command that brings the backup copy of the source VM's disk file found in the local mirror into the actual NFS share and gives it a new name.

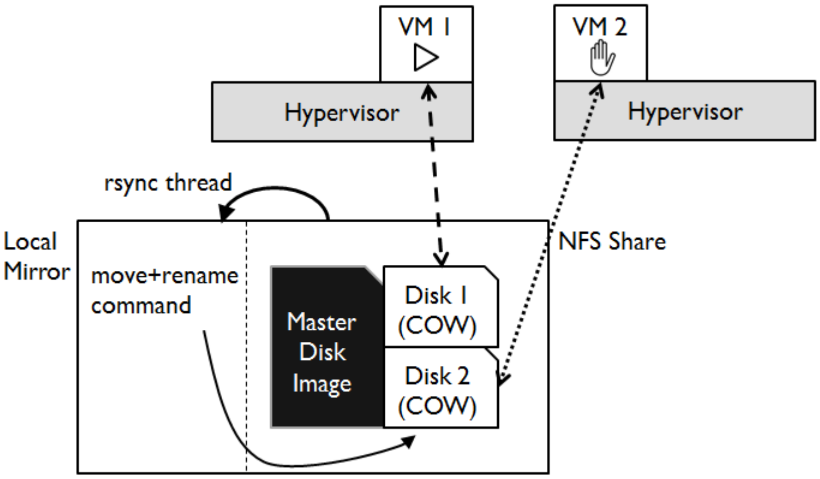


Figure 3.5: Pre-cloning preparations.

With the clone VM's storage correctly configured, the pre-copy algorithm can begin. Figure 3.6 portrays this phase of the cloning process, during which main memory consistency is gradually attained. The fact that the source VM is still running at the time means that its disk file's data may change as I/O write buffers get flushed by the VM's operating system. For this reason, a separate rsync thread is spawned to constantly synchronize the clone VM's disk file with that of the source VM.

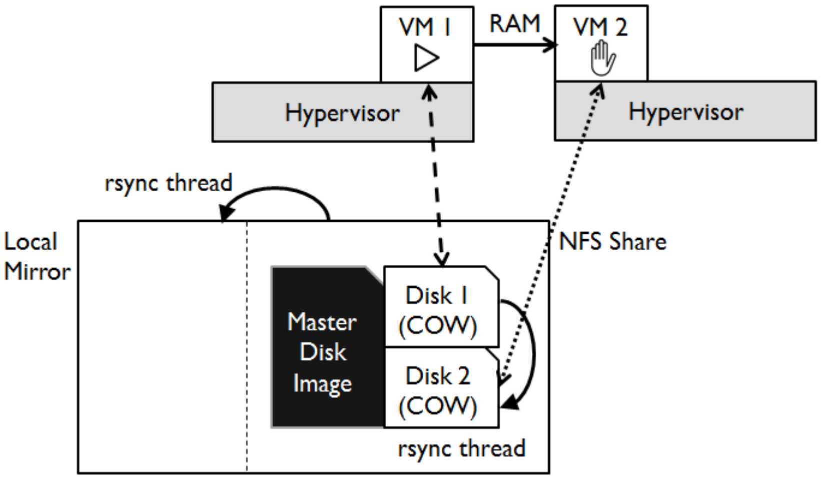


Figure 3.6: Storage state during memory page transfer.

The last pre-copy iteration entails pausing the source VM to transfer what is known as the *writeable working set (WWS)* or *hot pages*, which is a small, but frequently updated set of pages. We take advantage of this stop-and-copy phase that is inherent to the pre-copy algorithm to also ensure secondary storage consistency. As can be seen in Figure 3.7, we kill the rsync thread that was previously spawned and execute a final rsync instruction to update the clone VM's disk file with all of the changes exhibited by that of the source VM. Both VMs remain paused until this final synchronization completes.

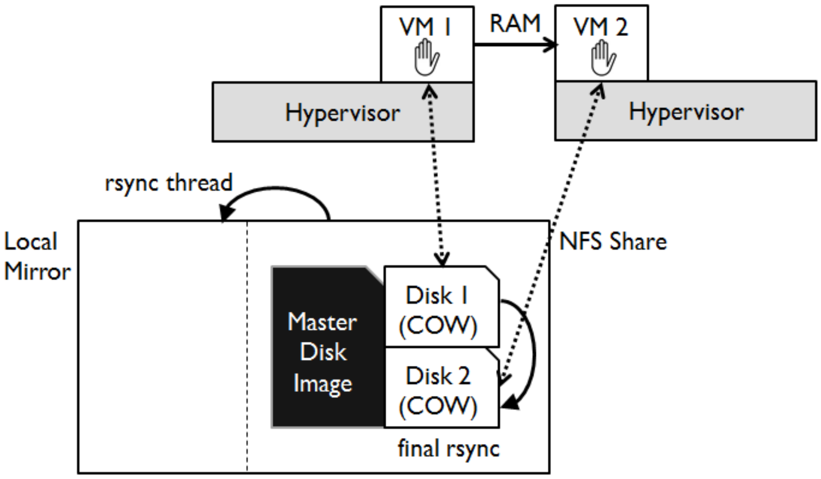


Figure 3.7: Storage operations on last pre-copy iteration.

The final step in the cloning operation is to resume both the source VM and its clone (see Figure 3.8). They now have completely independent disk files, so any updates to their filesystems will not affect the other VM. Full consistency between each VM's view of the filesystem and their associated disk file's state has been achieved. If the source VM's I/O buffers are not empty before the stop-and-copy phase, the clone VM and source VM will eventually commit those writes to their corresponding filesystems once they are resumed. Therefore, both disk files will end up reflecting the appropriate changes needed to preserve the aforementioned consistency. It should also be noted that the local mirror will start obtaining a new copy of the source VM's disk file, as well as a copy of the clone VM's disk file. These can be used for future cloning operations.

Pwnetizer generalizes what has been discussed in this subsection to clone VMs attached to N different disk files. This is easily accomplished by executing N move+rename commands mimicking the one in Figure 3.5, spawning N rsync threads inside the NFS share like the one in Figure 3.6, and invoking N final synchronizations such as the one in Figure 3.7.

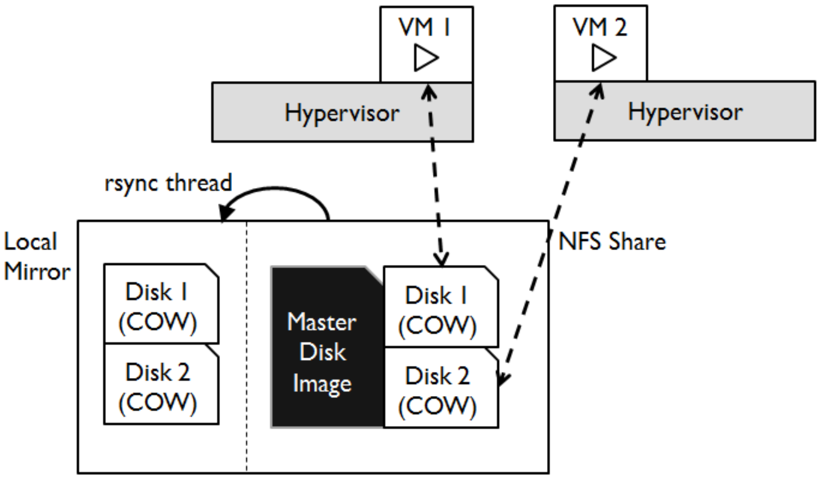


Figure 3.8: Storage state after cloning operation.

Host's Persistent Storage

Section 3.5.2 talks about changes that must be made to the runtime domain object found in the Hypervisor's main memory for the clone VM to run properly. These are modifications to the domain's name and UUID, along with new MAC addresses for the network cards and new secondary storage disk files. Those changes must also be written to a persistent medium for them to still be present next time the clone VM is launched. As mentioned in Section 3.5.1, Libvirt declares domains in a Hypervisor-agnostic XML format, so the Pwnetizer component's task in this respect is to translate the new domain object into an XML description compliant with Libvirt's XML schema and write it to the host's filesystem for later use.

3.5.4 Networking

VM Cloning requires runtime changes inside the cloned VM in order to avoid networking conflicts. There are three things that need to be different between the source VM and the target VM: MAC addresses, IP addresses, and hostnames. While MAC addresses are a physical property of the network cards and can be altered by the Hypervisor (see Section 3.5.2 and Section 3.5.3), IP addresses and hostnames are part of the VM's OS configuration. In addition, hardware features that are expected to remain the same (e.g., motherboard serial number, CPU model) are detected by the VM's operating system during boot-up and are not monitored thereafter; MAC addresses are one of these features. Thus, changes to the VM's internal state are also required if MAC address changes are to be reflected during runtime.

The clone VM must somehow detect that it has come alive (see Section 3.5.5) and immediately power off its network interface. That way, ongoing sessions and buffered packets are invalidated by the clone VM and are left for the original VM to handle without any interference. This action must be followed by swift modifications in the clone VM's configuration so that it can assume its new network identity and resume normal operation as soon as possible. In a typical Linux distribution, the steps to be followed by the clone VM are:

1. Turn off all network interfaces.
2. Modify any static IP configurations found in **/etc/network/interfaces** to adopt unused IP addresses and edit **/etc/hostname** to give the VM a unique hostname. These changes ensure that the new hostname and IP addresses become persistent in case the clone VM is rebooted some time in the future.
3. Use **hostname** and **ifconfig** commands to force the OS to update the VM's hostname and MAC addresses during runtime.
4. Turn all network interfaces back on. This will trigger DHCP requests if non-static IP configurations are being used by the VM.

Our Pwnetizer prototype runs a daemon inside every VM that detects the end of a cloning process (refer to Section 3.5.5) and takes care of all of these steps on the clone VM's side. This daemon is compatible with most popular Linux distributions (i.e., Fedora, Ubuntu, Gentoo, Debian, OpenSuSE, Mandrake) and is written in Java to avoid having to recompile its code on every platform.

3.5.5 Detecting Cloning

We need to find a way for the clone VM to realize that cloning has happened in order for it to begin reconfiguring its network settings. Even though virtualization infrastructure attempts to provide guest VMs with the illusion that they are running on top of physical hardware, researchers have discovered various methods to detect the presence of a Hypervisor from within a VM. For instance, Ferrie [19] [20] describes techniques of determining if an OS is running on top of VMWare, Bochs, Xen, QEMU. Unfortunately, we face a greater challenge: not only do we need to detect virtualization, we also need a VM to determine whether it is the clone VM or the original VM after a cloning operation. Only the clone VM should change its network settings; the original VM's configuration should remain unchanged. This means that we have to find some property that allows us to tell the underlying hosts apart. If the underlying host has changed for a VM, this would indicate that it is the clone VM instead of the original one.

VM Introspection

There are very limited ways for a VM to tell itself apart from others. In KVM, a VM's UUID is accessible from inside the VM through a special file (**/sys/devices/virtual/dmi/id/product_uuid**). However, an update to the VM's UUID is not reflected by the special file until the next time the VM is booted up, so this introspection mechanism does not meet our needs.

Hypercalls

A viable option to tell two VMs apart would be to take advantage of *hypercalls*. Hypercalls are Hypervisor-specific (e.g., Xen, KVM) and provide a procedural interface through which guest VMs can directly interact with the Hypervisor. By way of a hypercall, a VM could obtain information about the Hypervisor instance running beneath it and establish the underlying host's identity. However, this would require a virtualization-aware OS and would not be easily transferable from KVM to other virtualization platforms.

External Changes

Changes in the VM's surrounding environment may be easier to detect. One could envision the possibility of querying the network topology to find out whether or not a VM has changed its vantage point from inside the network. If so, that specific VM must be a clone VM. The main drawback of network-based detection methods comes from the fact that the original VM and the clone VM have the same network identity during the detection phase, which means that the clone VM could take over the forwarding rules associated with the original VM if it sends ICMP, ARP, or any other type of TCP/IP packets to the external network. This is attributed to the self-learning nature of ethernet switches, which update their forwarding tables as new packets pass through them.

For our Pwnetizer prototype, we employ our own network-based detection strategy that does not require packets to leave the host, thus minimizing the possibility of the original VM's network connectivity being disrupted by the clone VM before it changes its MAC and IP addresses. In a Cloud Computing scenario, a VM's network cards usually have a direct bridge to the host's physical Network Interface Card (NIC), which looks like the **eth1-br1-eth1** arrangement in Figure 3.9. This allows the VMs to be reachable from the outside because they are directly connected to the LAN, which is necessary when running any type of networked service based on incoming connections (e.g., mail server, web server, database server). As shown in Figure 3.9, we add a virtual Network Address Translation (NAT) gateway, **virbr0**, that runs inside each host and is presented to every

VM on a new NIC (i.e., **eth0**). While the fully-bridged interface (i.e., **eth1**) is used by the VMs to communicate with the external network, the virtual NAT gateway is used for the sole purpose of identifying the underlying host. Taking into account that each host presents a NAT gateway with the same IP address (i.e., **192.168.101.1**) and a different MAC address, the guest VMs can easily detect if they have switched hosts by querying **virbr0**'s MAC address. With this addition, each VM needs to store the NAT gateway's MAC address that is detected when the VM is powered on, followed by periodically checking if that MAC address has changed to determine whether or not a cloning operation has taken place. Since the original VM remains on the same host, no changes will be detected. On the other hand, the clone VM will recognize a different MAC address on the virtual NAT gateway, so the clone VM knows that its networking configuration must be updated as described in Section 3.5.4. These tasks are carried out by PwnetizerClient, an application-space program running inside the VM (see Section 3.5.6). It should be noted that querying the NAT gateway's MAC address is done by sending an ARP request containing **virbr0**'s IP address (**192.168.101.1**) to the **192.168.101.0/24** network segment, so the query packet never leaves the VM's host[2]. Consequently, this strategy is fast (small round-trip-times) and does not generate networking conflicts in the external network.

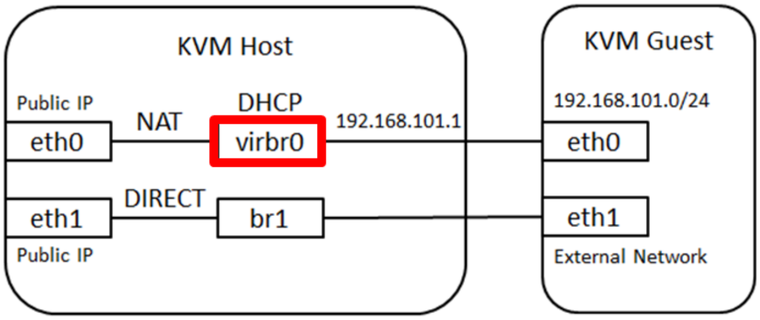


Figure 3.9: Prototype's Internal Networking.

3.5.6 End Product

Our KVM-based Pwnetizer prototype is comprised of three software components highlighted in Figure 3.10, for a total of 3,602 lines of code (LOC):

- **PwnetizerLibvirt** – a modified Libvirt that runs on each host and orchestrates the entire cloning procedure. It required an extra 2,362 lines of C code on top of Libvirt's original 484,349 LOC.

[2] The virtualized **192.168.101.0/24** network segment is entirely contained within each host.

- **PwnetizerServer** - manages the networked share where VM disk files are stored and takes care of efficient disk cloning. It is materialized in 1,109 lines of C code. Refer to Section 3.5.3 for details about its core tasks.

- **PwnetizerClient** - runs inside every VM and takes care of cloning detection as well as network reconfiguration. It is a small program, taking up only 131 lines of Java Code. Refer to Sections 3.5.4 and 3.5.5 for details about its core tasks.

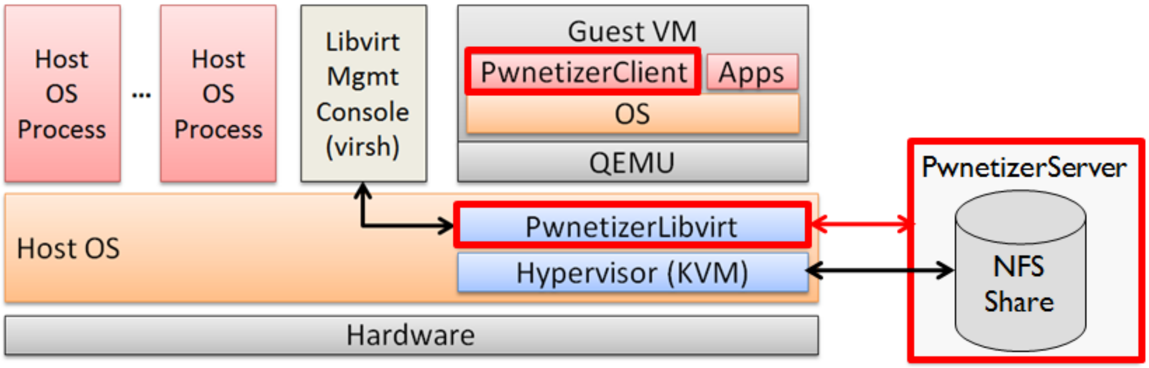


Figure 3.10: Pwnetizer Prototype Architecture.

Figure 3.11 summarizes all of the interactions that take place during a Pwnetizer VM Cloning procedure. The cloning action is triggered on the source Hypervisor instance, where the original VM is running. A full XML description of the VM (e.g., RAM size, number of vCPUs, NICs and their MAC addresses, etc) is generated and sent to the target Hypervisor, where the clone VM will reside. Once the target Hypervisor is done modifying the VM's definition to ensure that the clone VM is unique and does not interfere with the original VM's persistent storage or network, it contacts the PwnetizerServer application and gives it a list of disk files that will have to be cloned. Based on this information, PwnetizerServer prepares the clone VM's disk files and makes them available in the NFS share. With the new disk files now present in the NFS share, the target Hypervisor is able to launch the clone VM in suspended state and becomes ready for the pre-copy main memory page transfer to begin.

The pre-copy algorithm iteratively transfers main memory pages from the original VM to the clone VM while the original VM is powered on. When the time comes for the stop-and-copy phase, both Hypervisors notify PwnetizerServer and the original VM is paused. PwnetizerServer then proceeds to kill the synchronization threads between the original VM's and the clone VM's disk files, and issues a final synchronization call between them. As soon as the final synchronization is completed, both VMs have fully-consistent main memory and secondary storage states, so it is safe to power them back on, which is what happens when PwnetizerServer's **DONE** message reaches the two Hypervisor instances.

With both VMs now running, the PwnetizerClient application located inside the clone VM will detect a change in the host's virtual NAT gateway's MAC address (see Section 3.5.5), which makes it realize that it is dealing with the clone VM. As a result, the clone VM's PwnetizerClient proceeds to turn off all network interfaces, update the VM's network configuration to avoid connectivity issues with the original VM, and bring all network interfaces back online. Once this is done, both VMs are fully independent from each other and can be used to provide *crash resilience* and *micro-elasticity* for the cloud customer's applications and data, as mentioned in Section 3.1.1.

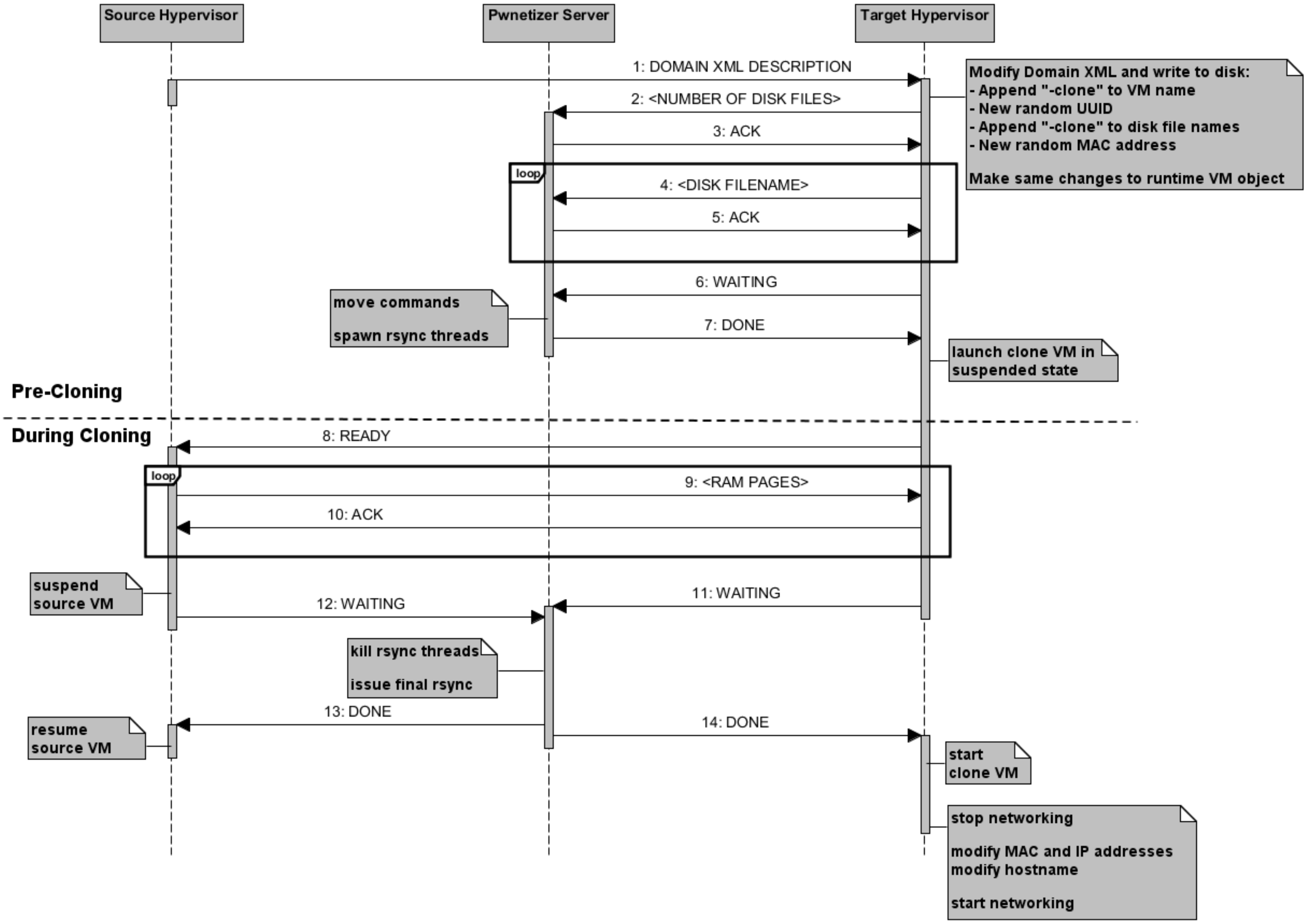


Figure 3.11: Pwnetizer Sequence Diagram.

3.6 Pwnetizer - OpenStack Deployment

Our KVM-based Pwnetizer prototype only offers the VM management facilities that come with Libvirt. These are sufficient for a small private cloud, but do not satisfy the scalability needs and usage scenarios of real public and private clouds. Furthermore, this limitation leads to burdensome host configuration and requires a lot of user expertise for it to be used. To improve Pwnetizer's usability, we decided to make it part of OpenStack [47], a full-blown open-source cloud environment that was launched in 2010 as a joint effort between Rackspace and NASA. OpenStack is now included in many popular Linux distributions, including Ubuntu, Fedora and OpenSUSE, and is actively being used to power large cloud services for big-name companies like CERN, NASA, HP Public Cloud and AT&T. When it comes to academic work, we believe that integrating Pwnetizer with OpenStack enables future research in Cloud Computing security. Princeton's PALMS research group has picked OpenStack as a unified platform for multiple security-related projects and has obtained satisfactory results.

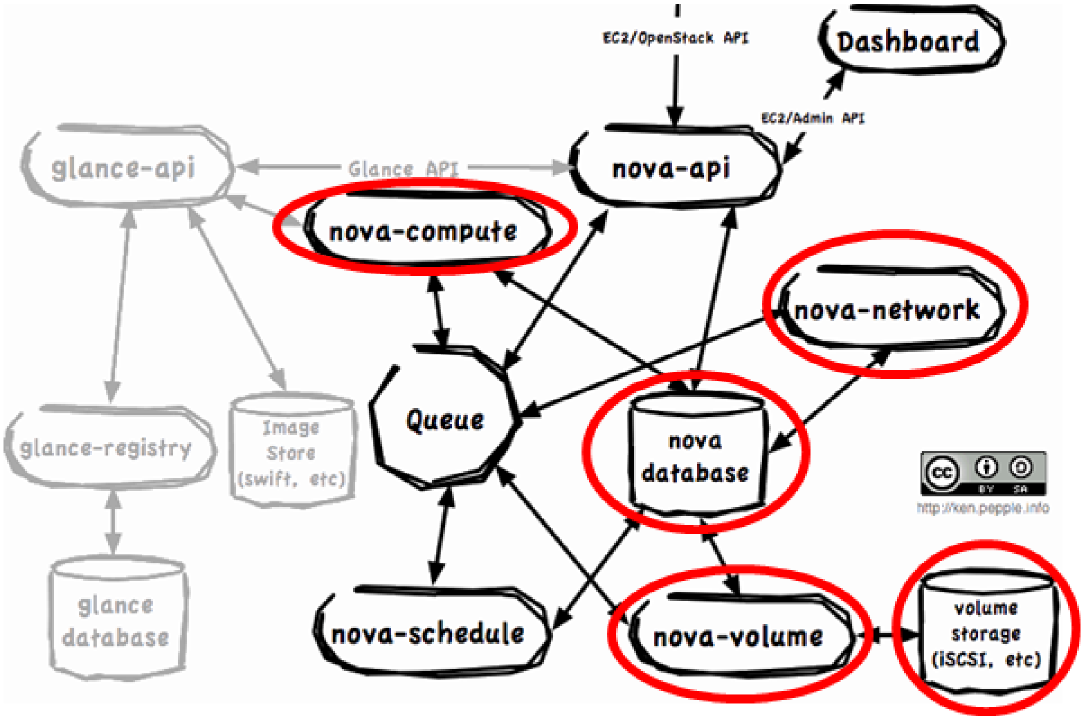


Figure 3.12: OpenStack Architecture.

Pwnetizer's OpenStack prototype required an extra 194 lines of Python code on top of OpenStack's original 151,227 LOC. Figure 3.12 highlights all OpenStack components that were affected by those changes. The modifications made are summarized below:

- **Nova-Compute**: A Nova-Compute instance manages every host in an OpenStack cloud. Nova-Compute delegates much of its functionality to Libvirt. Therefore, besides having to change every host's Libvirt to PwnetizerLibvirt, the Nova-Compute code was extended to contemplate the cloning scenario and handle it properly.

- **Nova DB**: The Nova DB stores information about the state of the cloud (e.g., VM instances, DHCP leases to VMs, disk volumes, resources available on each host). Once a cloning operation has successfully completed, a new VM instance corresponding to the clone VM is added to Nova DB, along with information about its disk volumes.

- **Nova-Volume and Volume Storage**: Nova-Volume and Volume Storage manage the networked file shares that contain all of the VM's disk files. For this reason, a PwnetizerServer must run alongside them to handle disk cloning operations related to VM Cloning. Our modified Nova-Volume makes sure to register every cloned volume appropriately.

- **Nova-Network**: The most challenging modifications were those made to Nova-Network, which is the component in charge of setting up firewall rules and allocating both private and public IP addresses to VM instances. After a cloning operation is finished, Nova-Network must allocate a new set of IP addresses for the clone VM. The DHCP entries related to those IP addresses must be tied to randomly-generated MAC addresses corresponding to the clone VM's virtual interfaces.

Figure 3.13 summarizes all of the interactions that take place during a VM Cloning procedure inside our Pwnetizer OpenStack prototype. It all begins when a **CLONE** command is issued through Nova-API, which can be done by either the cloud provider or the cloud customer. This command reaches the *source Nova-Compute*, where the original VM is currently located. In turn, the *source Nova-Compute* notifies the *target Nova-Compute*, which is running on top of the host where the clone VM will be generated. Once network filtering rules are set on the target side to allow for the VM's main memory pages to be transferred, both Nova-Compute instances communicate with their corresponding PwnetizerLibvirt to initiate a cloning procedure that is nearly identical to the one described in Section 3.5.6. The main variation is that, in this case, PwnetizerServer provides the target Hypervisor with a clone VM name generated based on the next available *VM id* so that it complies with OpenStack naming conventions. Throughout the *pre-cloning* and *during cloning* stages, the two Nova-Compute instances are just waiting while the Hypervisors perform the actual cloning.

The *post-cloning* phase is where most differences between this prototype and the previous one can be found. Once the Hypervisors have done their part, PwnetizerServer adds the clone VM's information to Nova DB and signals the *source Nova-Compute* to wake up by setting the original VM's state to **ACTIVE**. This re-activates the *source Nova-Compute*, who notifies the *target Nova-Compute* that it is time to make the finishing touches. Subsequently, the *target Nova-Compute* takes care of asking Nova-Network to allocate all the network resources required by the clone VM (i.e., private and public IP addresses, random MAC address, and firewall rules). Nova-Network promptly updates Nova DB with the resulting allocations, so the *target Nova-Compute* is able to find out what MAC address Nova-Network expects the Clone VM to have in order to fulfill its DHCP requests. To trigger the necessary network reconfiguration on the clone VM, the *target Nova-Compute* sends a *broadcast* packet indicating the (1) MAC address of the original VM along with the (2) MAC address that the clone VM must acquire. This packet is received by PwnetizerClient instances running inside every VM, which check if their VM's current MAC address matches the MAC address of the VM that has just been cloned. If so, they query the virtual NAT gateway's MAC address to check whether it has changed or not (see Section 3.5.5). Only the PwnetizerClient running inside the clone VM will notice a change in the virtual NAT gateway's MAC address, so it updates the VM's MAC address to the one contained in the broadcast packet and asks for a new DHCP lease on behalf of the clone VM. Meanwhile, the PwnetizerClient running inside the original VM sends a series of ping packets to an arbitrary external server in order to safeguard the network's forwarding rules corresponding to the original VM, as they may be hijacked by the clone VM before it acquires its new MAC and IP addresses. This minor tweak is necessary because of the self-learning nature of ethernet switches, which update their forwarding tables as new packets pass through them.

Once the *post-cloning* phase is over, both VMs are fully independent from each other and can be used to provide *crash resilience* and *micro-elasticity* for the cloud customer's applications and data, as mentioned in Section 3.1.1.

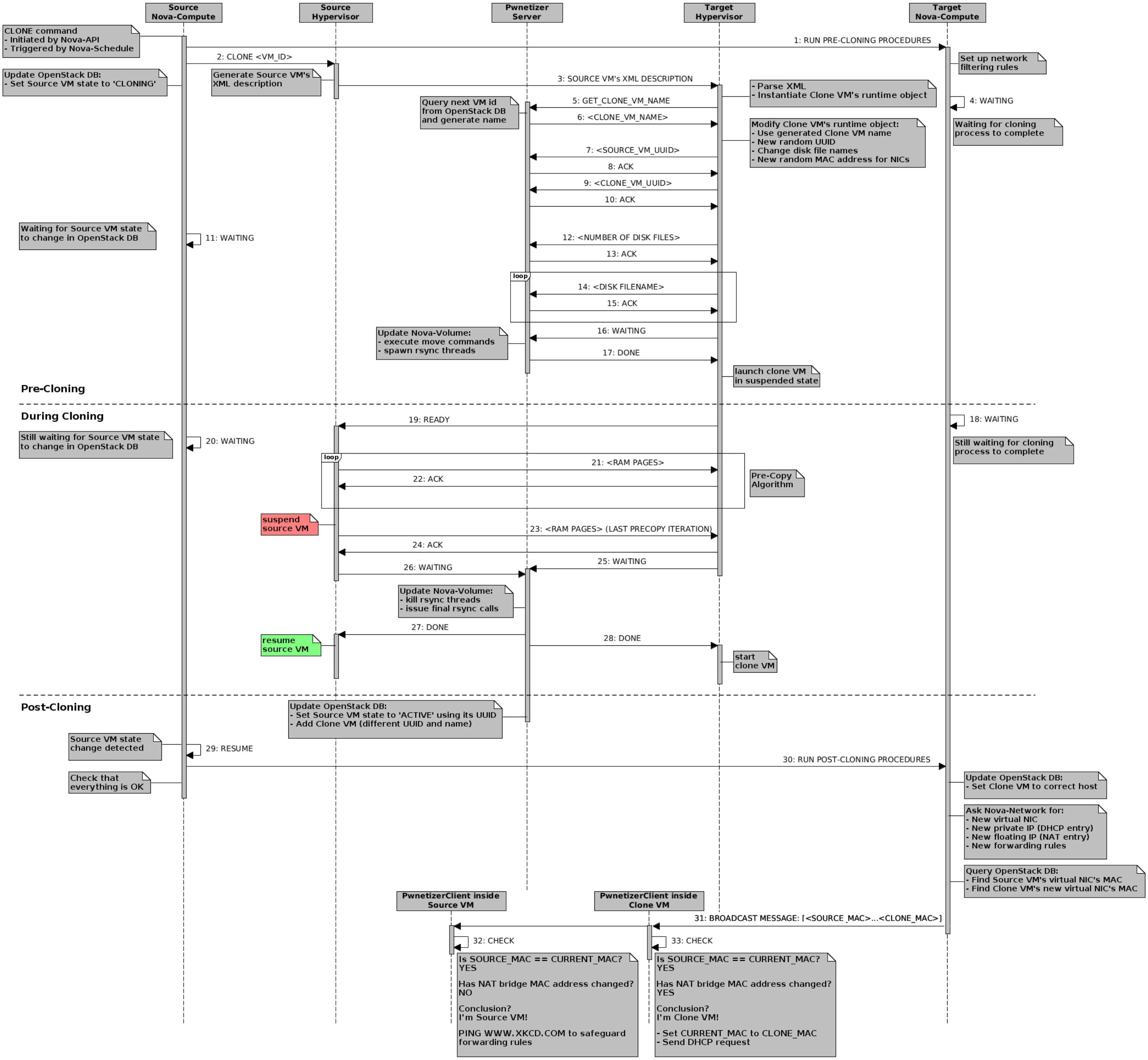

Source Nova-Compute
Source Hypervisor
Pwnetizer Server
Target Hypervisor
Target Nova-Compute
CLONE command
- Initiated by Nova-API
- Triggered by Nova-Schedule
1: RUN PRE-CLONING PROCEDURES
Set up network filtering rules
2: CLONE <VM_ID>
Update OpenStack DB:
- Set Source VM state to 'CLONING'
Generate Source VM's XML description
3: SOURCE VM's XML DESCRIPTION
- Parse XML
- Instantiate Clone VM's runtime object
4: WAITING
Query next VM id from OpenStack DB and generate name
5: GET_CLONE_VM_NAME
6: <CLONE_VM_NAME>
Modify Clone VM's runtime object:
- Use generated Clone VM name
- New random UUID
- Change disk file names
- New random MAC address for NICs
Waiting for cloning process to complete
7: <SOURCE_VM_UUID>
8: ACK
9: <CLONE_VM_UUID>
10: ACK
Waiting for Source VM state to change in OpenStack DB
11: WAITING
12: <NUMBER OF DISK FILES>
13: ACK
loop
14: <DISK FILENAME>
15: ACK
16: WAITING
Update Nova-Volume:
- execute move commands
- spawn rsync threads
17: DONE
launch clone VM in suspended state
Pre-Cloning
During Cloning
19: READY
18: WAITING
Still waiting for Source VM state to change in OpenStack DB
20: WAITING
Still waiting for cloning process to complete
loop
21: <RAM PAGES>
22: ACK
Pre-Copy Algorithm
suspend source VM
23: <RAM PAGES> (LAST PRECOPY ITERATION)
24: ACK
25: WAITING
26: WAITING
Update Nova-Volume:
- kill rsync threads
- issue final rsync calls
27: DONE
resume source VM
28: DONE
start clone VM
Post-Cloning
Update OpenStack DB:
- Set Source VM state to 'ACTIVE' using its UUID
- Add Clone VM (different UUID and name)
Source VM state change detected
29: RESUME
30: RUN POST-CLONING PROCEDURES
Check that everything is OK
Update OpenStack DB:
- Set Clone VM to correct host
Ask Nova-Network for:
- New virtual NIC
- New private IP (DHCP entry)
- New floating IP (NAT entry)
- New forwarding rules
Query OpenStack DB:
- Find Source VM's virtual NIC's MAC
- Find Clone VM's new virtual NIC's MAC
PwnetizerClient inside Source VM
PwnetizerClient inside Clone VM
31: BROADCAST MESSAGE: [<SOURCE_MAC>...<CLONE_MAC>]
32: CHECK
33: CHECK
Is SOURCE_MAC == CURRENT_MAC?
YES
Has NAT bridge MAC address changed?
NO
Conclusion?
I'm Source VM!
PING WWW.XKCD.COM to safeguard forwarding rules
Is SOURCE_MAC == CURRENT_MAC?
YES
Has NAT bridge MAC address changed?
YES
Conclusion?
I'm Clone VM!
- Set CURRENT_MAC to CLONE_MAC
- Send DHCP request

Figure 3.13: Sequence diagram of Pwnetizer's OpenStack implementation.

3.7 Pwnetizer - Optimizations

Minimizing VM Cloning downtime and total time were high-priority objectives. For this reason, we had to tweak the Pwnetizer OpenStack prototype as much as possible to favor those two metrics. We now give three examples of optimizations that helped us achieve improved performance.

3.7.1 No gratuitous ARP packet after cloning

Pwnetizer uses KVM's live migration module as a building block. One minor detail that has serious implications is that, after migration, it is common practice for a gratuitous ARP packet to be broadcast on the network to redirect network packets designated to the IP address of the migrated VM from the source host to the destination host. However, in the case of VM Cloning, this action leads to the original VM's network identity being hijacked by the clone VM. Even though we could change Pwnetizer so that the original VM was the one in charge of acquiring a new MAC and IP address, it is wasteful to force an update in the LAN's forwarding rules if there is no need to do that in the first place.

We found that not sending the gratuitous ARP packet after cloning led to significantly faster network convergence because the forwarding rules associated with the original VM were preserved, so the clone VM's network reconfiguration procedure did not require traffic to switch back to the original VM. This was accomplished by changing a single line of code in KVM.

3.7.2 Tweaking process priorities

In a normal OpenStack deployment, both the NFS share and the *controller node* are installed on the same host. A *controller node* orchestrates an entire cloud and runs a variety of services, including the main Nova-Network service, which is in charge of managing all of the DHCP leases. There exists a single *controller node* in an OpenStack cloud; the rest of the hosts act as *compute nodes.* During our initial VM Cloning experiments, we realized that PwnetizerServer's synchronization threads that maintain the NFS share's local mirror (see Section 3.5.3) interfere with time-critical tasks during Pwnetizer's *post-cloning* phase (see Section 3.6). Once a cloning procedure is underway, copies of the original VM's disk files are moved from the local mirror into the NFS share (see Section 3.5.3), so the synchronization threads will attempt to re-generate them on the local mirror right away. Unfortunately, this non-critical task will take up CPU time from bottleneck tasks, such as allocating the clone VM's network resources through Nova-Network.

Linux schedulers take *niceness* values into account when deciding how much CPU time to allocate to each running process. Although it may be counter-intuitive, niceness values of -20 indicate the highest priority and values of 19 translate into the lowest priority. Niceness values of 0 are the default for any process. By assigning niceness values of -20 to all OpenStack processes and setting PwnetizerServer's niceness value to 19, we observed an improvement of approximately 30% in *post-cloning* times, which stands for a 10-second improvement in total cloning times.

3.7.3 Tweaking KVM's pre-copy settings

KVM's live migration code, which is at the heart of Pwnetizer's VM Cloning prototype, has two problematic default values: the maximum network bandwidth to be used is set to 32 Mbps and the target downtime of the process is set to 30 ms. Imposing a bandwidth limit to page transfers results in more pre-copy iterations and longer cloning times than could be achieved by taking full advantage of the host's network resources. Meanwhile, setting the downtime goal to a very small value (i.e., 30 ms) forces the pre-copy algorithm to continue iterating until the *writeable working set (WWS)* is reduced to a tiny fraction of the VM's total memory, which can take extended periods of time and might not even happen in some cases – the result being a never-ending cloning procedure in the worst case. Fortunately, Libvirt's API provides two commands (**migrate-setspeed** and **migrate-setmaxdowntime**) that overwrite those values once a migration operation is ongoing. In our Pwnetizer prototype, we execute those commands as soon as cloning begins. We instruct KVM to use up as much bandwidth as possible and to aim for a 500 ms downtime, which we found to be an optimal trade-off between downtime and total cloning time. These subtle changes gave us an average time reduction of approximately 300% in the *during cloning* phase, which stands for a 30-second improvement in total cloning times.

3.8 Pwnetizer - Results

Up until now, we have described the motivation and design decisions behind our Pwnetizer VM Cloning technique, as well as some of the prototype's inner workings. To evaluate the degree to which we have accomplished our objectives, we now run detailed benchmarks on the Pwnetizer OpenStack prototype and conduct a series of experiments with popular network-intensive and computationally-intensive applications.

Test Bed

Our test bed is comprised of two hosts with identical hardware and software configurations. Each host runs Ubuntu 12.04 LTS with kernel version 2.6.38.8 and comes with dual quad-core Intel Nehalem CPUs (1.6GHz) as well as 6 GB of RAM. The network connecting both hosts supports 1Gbps speeds. Meanwhile, the VMs being cloned are m1.small instances with 2 GB of dedicated RAM and 1 vCPU core. All VMs run Ubuntu 12.04 LTS Server Edition.

Just like in a normal OpenStack deployment, one host runs both the NFS share and the *controller node*, while the other one acts as a simple *compute node*. Refer to Section 3.7.2 to learn about the implications of this arrangement.

3.8.1 Fine-Grained Benchmarking

For us to better understand the performance characteristics of our VM Cloning strategy, we break down the entire process into 17 different stages and characterize their behavior with several standard Cloud Computing-specific workloads.

Considering that our Pwnetizer OpenStack code is comprised of more than 4 distinct software modules (i.e., PwnetizerLibvirt, PwnetizerClient, PwnetizerServer, modified Nova-Compute) distributed over multiple hosts, we decided to simplify benchmarking by storing time measurements inside a centralized SQL database. Every module connects to the database and adds simple [event_name, timestamp] tuples whenever a new cloning stage begins or ends. This scheme eliminates the need to synchronize the clocks of the various hosts, since all SQL updates can use SQL's **current_timestamp** keyword to generate timestamps based on the database server's system clock.

High-Level Metrics

Figure 3.14 shows the various stages that we identified in our prototype. Appendix B provides a short description for each stage.

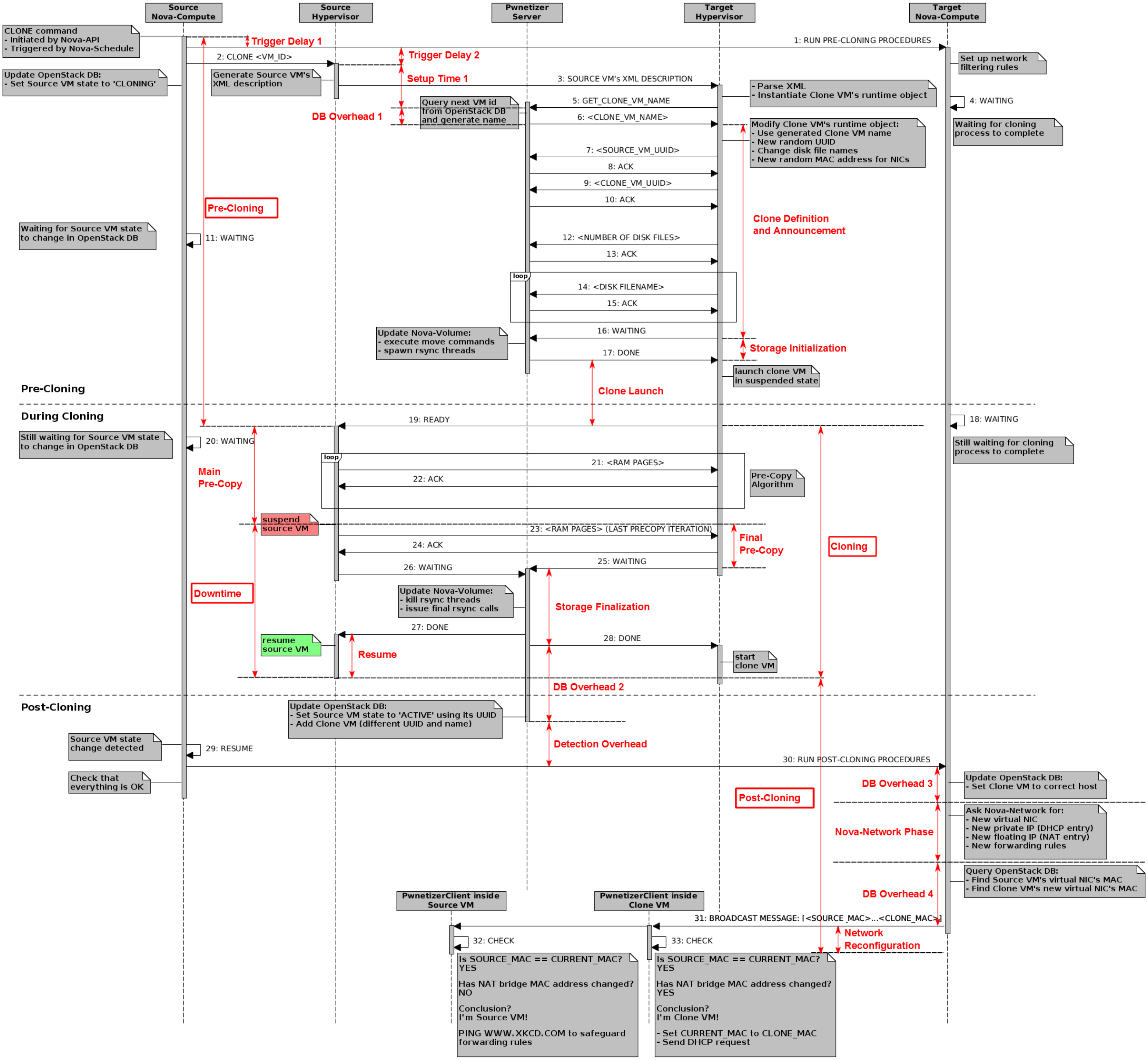

Source Nova-Compute
Source Hypervisor
Pwnetizer Server
Target Hypervisor
Target Nova-Compute
CLONE command
- Initiated by Nova-API
- Triggered by Nova-Schedule
Trigger Delay 1
1: RUN PRE-CLONING PROCEDURES
2: CLONE <VM_ID>
Trigger Delay 2
Set up network filtering rules
Update OpenStack DB:
- Set Source VM state to 'CLONING'
Generate Source VM's XML description
Setup Time 1
3: SOURCE VM's XML DESCRIPTION
- Parse XML
- Instantiate Clone VM's runtime object
Query next VM id from OpenStack DB and generate name
5: GET_CLONE_VM_NAME
4: WAITING
DB Overhead 1
6: <CLONE_VM_NAME>
Modify Clone VM's runtime object:
- Use generated Clone VM name
- New random UUID
- Change disk file names
- New random MAC address for NICs
Waiting for cloning process to complete
7: <SOURCE_VM_UUID>
8: ACK
9: <CLONE_VM_UUID>
10: ACK
Pre-Cloning
Clone Definition and Announcement
Waiting for Source VM state to change in OpenStack DB
11: WAITING
12: <NUMBER OF DISK FILES>
13: ACK
loop
14: <DISK FILENAME>
15: ACK
Update Nova-Volume:
- execute move commands
- spawn rsync threads
16: WAITING
Storage Initialization
17: DONE
launch clone VM in suspended state
Clone Launch
Pre-Cloning
During Cloning
19: READY
18: WAITING
Still waiting for Source VM state to change in OpenStack DB
20: WAITING
Still waiting for cloning process to complete
loop
21: <RAM PAGES>
Main Pre-Copy
22: ACK
Pre-Copy Algorithm
suspend source VM
23: <RAM PAGES> (LAST PRECOPY ITERATION)
Final Pre-Copy
Cloning
24: ACK
26: WAITING
25: WAITING
Downtime
Update Nova-Volume:
- kill rsync threads
- issue final rsync calls
Storage Finalization
27: DONE
resume source VM
28: DONE
Resume
start clone VM
DB Overhead 2
Post-Cloning
Update OpenStack DB:
- Set Source VM state to 'ACTIVE' using its UUID
- Add Clone VM (different UUID and name)
Source VM state change detected
Detection Overhead
29: RESUME
30: RUN POST-CLONING PROCEDURES
Check that everything is OK
DB Overhead 3
Update OpenStack DB:
- Set Clone VM to correct host
Post-Cloning
Ask Nova-Network for:
- New virtual NIC
- New private IP (DHCP entry)
- New floating IP (NAT entry)
- New forwarding rules
Nova-Network Phase
DB Overhead 4
Query OpenStack DB:
- Find Source VM's virtual NIC's MAC
- Find Clone VM's new virtual NIC's MAC
PwnetizerClient inside Source VM
PwnetizerClient inside Clone VM
31: BROADCAST MESSAGE: [<SOURCE_MAC>...<CLONE_MAC>]
Network Reconfiguration
32: CHECK
33: CHECK
Is SOURCE_MAC == CURRENT_MAC?
YES
Has NAT bridge MAC address changed?
NO
Conclusion?
I'm Source VM!
PING WWW.XKCD.COM to safeguard forwarding rules
Is SOURCE_MAC == CURRENT_MAC?
YES
Has NAT bridge MAC address changed?
YES
Conclusion?
I'm Clone VM!
- Set CURRENT_MAC to CLONE_MAC
- Send DHCP request

Figure 3.14: The different stages of Pwnetizer's OpenStack implementation overlaid on top of the corresponding sequence diagram.

From these stages, five high-level metrics can be calculated:

1. ***Pre-cloning* time** – Time taken to get ready for the main memory page transfer to begin. It starts with the **TRIGGER DELAY 1** phase and ends when the **MAIN PRE-COPY** stage begins.

2. ***During cloning* time** – Time interval during which main memory and secondary storage consistency is reached between the original VM and the clone VM. It begins with the **MAIN PRE-COPY** stage and ends when the **RESUME** stage is over.

3. ***Post-cloning* time** – Time taken by all the operations leading to a new network identity for the clone VM after it is powered on. It goes from the end of the **RESUME** stage until the completion of the **NETWORK RECONFIGURATION** stage.

4. **Downtime** – It is a measure of liveliness and corresponds to the time taken by the **FINAL PRE-COPY**, **STORAGE FINALIZATION** and **RESUME** stages, which is when incoming packets are lost and no computational tasks are scheduled due to the fact that both the original VM and the clone VM are paused.

5. **Total cloning time** – Time elapsed between the moment when the **CLONE** command is issued and the instant when the clone VM's network reconfiguration is over. It begins at the start of the **TRIGGER DELAY 1** phase and ends when the **NETWORK RECONFIGURATION** stage is over.

Typical Workloads

Applications can be divided into two categories: (1) those that are purely computational and (2) those that mix computation with disk I/O and/or network I/O. The latter are more common in a Cloud Computing setting, where 7 different types of workloads arise [27]. Table 3.1 lists those workloads and describes the benchmarks that we used to simulate them in a standardized way. For all client-server benchmarks, the VM was assigned the task of running the server-side process, which is what is expected from a VM deployed in the cloud.

Methodology

For each workload, we ran 20 trials. We then averaged those results to obtain a faithful idea of the overall metrics associated with each workload.

Table 3.1: Cloud Computing Workloads

Workload	**Benchmark**
Mail Server	mstone [43] as remote SMTP client; smtp-sink [56] as SMTP server inside VM
App Server	Faban Benchmarking Framework [18] as remote client; GlassFish Server [22] with sample Java EE application inside VM
File Server	Dbench [12] inside VM
Web Server	Faban Benchmarking Framework [18] as remote client; Apache HTTP Server [1] inside VM
DB Server	Sysbench [60] inside VM
Stream Server	VideoLAN [63] inside VM; Wireshark [69] capturing stream packets remotely
Idle Server	No workload

Performance Overview

Table 3.2 shows the five high-level metrics obtained with the various workloads. It can be seen that we achieve sub-second downtimes in most cases and experience total cloning times ranging between 11 and 33 seconds. Furthermore, the worst-case downtime of 1.10 seconds is still very close to the 1-second mark. These numbers are considerably better than those attained when employing CloneScale's [58] technique for creating fully-independent clone VMs, with downtimes in the order of tens of seconds and total times of at least 30 seconds.

Table 3.2: Summary of Pwnetizer VM Cloning performance numbers with typical Cloud Computing workloads.

Workload	**Downtime (s)**	**Pre-Cloning (s)**	**Cloning (s)**	**Post-Cloning (s)**	**Total (s)**
Mail Server	0.46	3.60	4.44	4.67	12.71
App Server	0.50	3.46	7.33	4.62	15.42
File Server	0.63	9.29	5.73	17.96	32.99
Web Server	0.90	7.31	6.57	11.41	25.29
DB Server	0.49	4.60	12.27	8.45	25.31
Stream Server	1.10	4.34	14.43	7.30	26.07
Idle Server	0.41	3.48	3.75	3.92	11.15
Average	**0.64**	**5.15**	**7.79**	**8.33**	**21.28**

The best-case scenario is the Idle Server – a VM without any processes running other than its basic OS services. Under those circumstances, the clone VM is fully-functional after only 11.15 seconds and the original VM experiences a negligible 410-millisecond downtime. When it comes to more realistic scenarios, the Mail Server and App Server workloads are very close to the ideal case, both with total cloning times below 13 seconds and downtimes of 500 milliseconds or less.

Pre-cloning and post-cloning times are mostly dependent on the disk cloning tasks. Therefore, it makes sense that the worst pre-cloning and post-cloning times are exhibited by the File Server workload, since it is the one issuing the largest amount of disk writes. This leads to longer waits and increased CPU load in OpenStack's *controller node* when synchronizing the original VM's and the clone VM's disk files in the NFS share. Given that synchronization threads attempt to re-generate the original VM's disk files in the local mirror as soon as the cloning operation is underway, they interfere with the *controller node*'s OpenStack-related tasks. These symptoms are aggravated if the NFS share is receiving large amounts of write requests, which is the case with the File Server workload.

Cloning times and downtimes are dominated by page dirtying inside the VM. It can be observed that the Stream Server generates the worst results in both of those metrics, which can be explained by the fact that video streaming requires *live transcoding* of video files. Video transcoding is the conversion of a previously-compressed video signal into another one with different characteristics (i.e., bit rate, frame rate, frame size, compression standard). Thus, the Stream Server generates the highest computational load amongst all workloads, which translates into a higher page dirtying rate. Consequently, the Stream Server requires more pre-copy iterations during the *cloning* stage, resulting in a longer **MAIN PRE-COPY** stage. The Stream Server also has a larger *writeable working set (WWS)*, leading to increased downtimes due to a longer stop-and-copy phase.

Detailed Performance Analysis

Table 3.3 shows the numbers obtained at a per-stage granularity (see Appendix B for a description of every stage). This allows us to identify the main bottleneck operations for VM Cloning. The first thing that can be noticed is that, out of the 17 stages, 13 of them consistently stay below the 1-second mark. The four events that govern the overall performance of our Pwnetizer OpenStack prototype are: **TRIGGER DELAY 2**, **CLONE LAUNCH**, **MAIN PRE-COPY**, and **NOVA-NETWORK**. Times associated with the **CLONE LAUNCH** phase do not raise any alarms, so we will concentrate on the other three bottleneck events.

The **MAIN PRE-COPY** stage takes care of transferring all main memory pages from the original VM to the clone VM, so the fact that it lasts several seconds is unsurprising. **MAIN PRE-COPY** times are a function of the size of the VM's main memory space (i.e., its allocated RAM), the workloads running inside the VM, and the available network bandwidth. Therefore, two ways to further decrease those times are: (1) using smaller VMs (e.g., 512MB instead of 2GB of RAM) and (2) increasing the network's bandwidth (e.g., 10Gbps instead of 1 Gbps).

TRIGGER DELAY 2 and **NOVA-NETWORK** have something in common: they both issue calls to a third-party application. While the former requires Nova-Compute to invoke a PwnetizerLibvirt command, the latter requires Nova-Network to interact with a third-party DHCP server

Table 3.3: Detailed Pwnetizer VM Cloning performance numbers with typical Cloud Computing workloads

Event Name	**Mail Server (s)**	**App Server (s)**	**File Server (s)**	**Web Server (s)**	**DB Server (s)**	**Stream Server (s)**	**Idle Server (s)**	**Avg. (s)**
TRIGGER DELAY 1	0.12	0.10	0.52	0.69	0.19	0.08	0.11	0.26
TRIGGER DELAY 2	1.52	1.45	3.85	2.54	1.81	1.82	1.47	2.07
SETUP TIME 1	0.06	0.06	0.82	1.27	0.15	0.53	0.07	0.42
DB OVERHEAD 1	0.05	0.04	0.69	0.09	0.07	0.04	0.05	0.15
CLONE DEF. & ANN.	0.41	0.43	0.96	0.84	0.53	0.40	0.41	0.57
STORAGE INIT.	0.12	0.10	0.57	0.53	0.22	0.11	0.11	0.25
CLONE LAUNCH	1.31	1.27	1.88	1.35	1.63	1.35	1.25	1.44
MAIN PRE-COPY	3.99	6.83	5.10	5.66	11.78	13.32	3.35	7.15
FINAL PRE-COPY	0.13	0.15	0.32	0.37	0.14	0.83	0.13	0.30
STORAGE FIN.	0.23	0.23	0.22	0.39	0.25	0.22	0.21	0.25
RESUME	0.09	0.12	0.09	0.14	0.10	0.06	0.07	0.09
DB OVERHEAD 2	0.44	0.41	1.09	0.72	0.60	0.22	0.22	0.53
DETECT. OVERHEAD	0.26	0.35	0.63	0.20	0.29	0.27	0.21	0.31
DB OVERHEAD 3	0.19	0.17	1.52	1.18	0.48	0.17	0.13	0.55
NOVA-NETWORK	3.46	3.34	13.95	8.30	6.31	6.25	3.00	6.37
DB OVERHEAD 4	0.11	0.07	0.50	0.80	0.38	0.05	0.05	0.28
NETWORK RECONF.	0.31	0.41	0.37	0.35	0.49	0.40	0.38	0.39
DOWNTIME	0.46	0.50	0.63	0.90	0.49	1.10	0.41	0.64
PRE-CLONING	3.60	3.46	9.29	7.31	4.60	4.34	3.48	5.15
DURING CLONING	4.44	7.33	5.73	6.57	12.27	14.43	3.75	7.79
POST-CLONING	4.67	4.62	17.96	11.41	8.45	7.30	3.92	8.33
TOTAL CLONING	12.71	15.42	32.99	25.29	25.31	26.07	11.15	21.28

(Dnsmasq [16]). The use of a Remote Procedural Call or some other inter-process communication mechanism explains **TRIGGER DELAY 2**'s two-second overhead. Nonetheless, **NOVA-NETWORK** takes up to 14 seconds in some cases and has an average time of 6.37 seconds, which is still very significant. We believe that Nova-Network has to carry out a considerable amount of work (i.e., set up new DHCP entries, update Nova DB, generate random MAC addresses, set up LAN-wide firewall rules) during the **NOVA-NETWORK** phase, so we still expect it to require more time than a typical cloning stage. To reduce Nova-Network's impact over the total cloning time, we recommend deploying the NFS share along with PwnetizerServer on a separate host to stop *disk cloning*-related processes from interfering with Nova-Network's tasks (see Section 3.7.2).

It should be noted that the **STORAGE FINALIZATION** stage always takes less than 400 milliseconds and has an average of 250 millisecomds across all workloads. This proves that our *disk cloning* procedure described in Section 3.5.3 successfully reduces *disk cloning*-related downtimes to the same order of magnitude as *page transfer*-related downtimes associated with traditional Live VM Migration.

3.8.2 Experimental Evaluation

The fine-grained benchmarking results showed us that our Pwnetizer OpenStack prototype can generate fully-independent clone VMs in a timely manner with negligible downtime regardless of the VM's internal workload. Now, we are interested in analyzing the effects that Pwnetizer VM Cloning can have on an application's performance (i.e., throughput, delay, running time). To this end, we study two common types of Cloud Computing applications: (1) a web application server (GlassFish Server [22]), and (2) a distributed computing framework (Cajo [7]). All load-balanced scenarios use an instance of HAProxy [24] between the client programs and the VMs.

Constant Web Load

Figure 3.15 shows the throughput numbers obtained when a VM running a GlassFish server is exposed to a constant web load generated by a remote Faban's [18] CoreHTTP client. The dotted line represents the case where no cloning is performed. Meanwhile, the continuous line corresponds to a trial in which cloning is initiated at the 50-second mark while load balancing is being performed between the original VM and the clone VM. Finally, the dashed line corresponds to a similar trial, but without load balancing between the VMs.

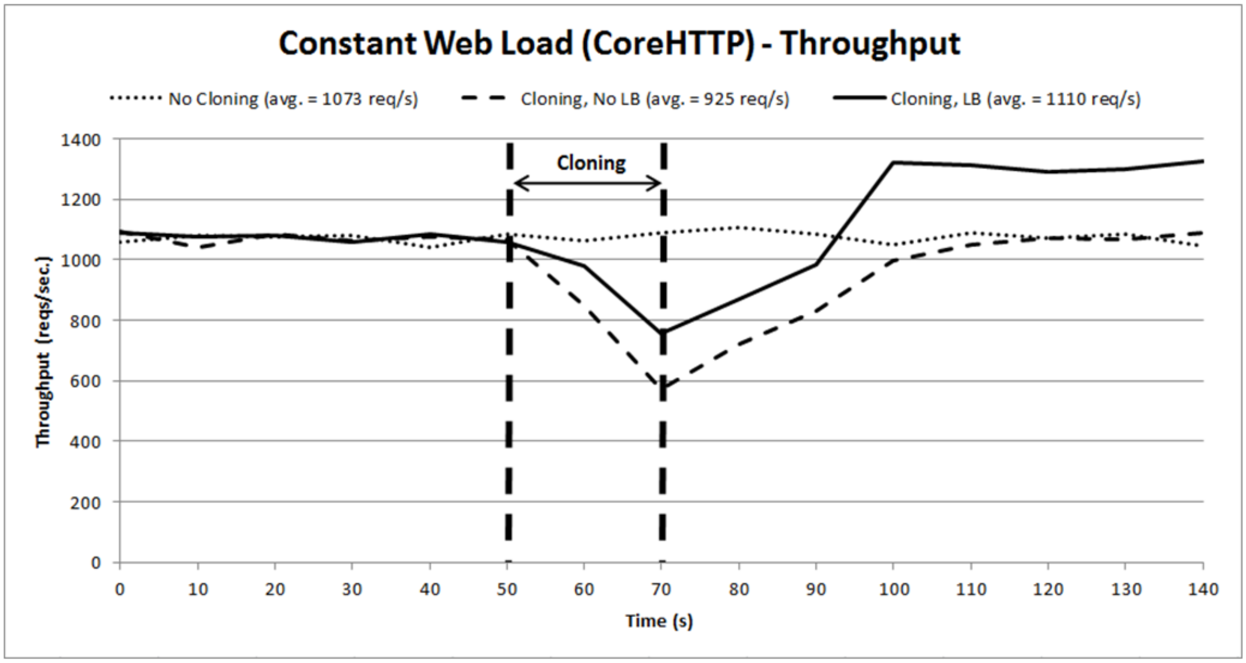


Figure 3.15: GlassFish Server throughput under constant web load.

Let us first analyze the application's throughput when cloning takes place under the absence of a load balancer. The initial throughput of approximately 1,073 requests per second (reqs/s) is the best that a single VM can provide. Between t=50s and t=70s, we observe the performance degradation caused by cloning. Once VM Cloning is done, throughput goes back to its original levels. It should be noted that no packet loss is observed. Over the course of 140 seconds, the average throughput is of 925 reqs/s as opposed to the 1073 reqs/s seen in the *No Cloning* scenario. Thus, the performance overhead in this case is of 13.8%.

A similar behavior is seen when we place a load balancer between the client and the VMs. There is a 20-second interval of degraded performance (t=50s to t=70s) because of the cloning operation followed by a swift throughput recovery. However, load balancing allows the application to provide a higher level of throughput than was possible with a single VM. While pre-cloning throughput was stable at 1,073 reqs/sec, the post-cloning throughput stabilized at approximately 1,300 reqs/sec, corresponding to a 21.2% improvement. Given that we now have two VMs handling web requests, it would be reasonable to naively expect double the throughput. In reality, the application's *capacity* has indeed doubled, but the client's web load has stayed the same. Therefore, the application's quality of service is only marginally improved by the presence of a clone VM because it helps decrease the time that web packets spend in a server's queue before being serviced. Since the VMs still have to wait for the client to actually send its web requests, throughput can only improve up to a certain point under constant web load. Over the course of 140 seconds, the average throughput in this case is of 1110 reqs/s, which is a 3.4% improvement over the 1073 reqs/sec of the *No Cloning* scenario. This indicates that, 90 seconds after cloning was triggered, the added throughput resulting from load balancing between the two VMs had already compensated for the performance cost inflicted by the cloning operation.

We now turn our attention to Figure 3.16, which shows the delays measured during the same experiments. The *No Cloning* scenario holds a stable 92-millisecond delay per request. On the other hand, the two trials that result in a clone VM exhibit a noticeable delay spike once the cloning operation starts at t=50s. The highest delays are below 200 ms, so they are still tolerable under most situations. After the clone VM is ready (t=70s), the delays start dropping back to normal levels. At the 100-second mark, the cloning scenario without a load balancer reaches its pre-cloning state. In contrast, the load-balanced VMs improve the application's quality of service by converging to a stable 74-millisecond delay – 19.6% less than pre-cloning delays. Over the course of the experiments, the cloning scenario without a load balancer reported an average delay of 104 ms, which is a 13.0% increase with respect to the *No Cloning* run. Meanwhile, the load-balanced trial had an average delay of 89 ms, which translates into a 3.3% improvement. Thus, as with throughput, the load-balanced VMs had already compensated for VM Cloning's performance impact just 90 seconds after cloning was triggered.

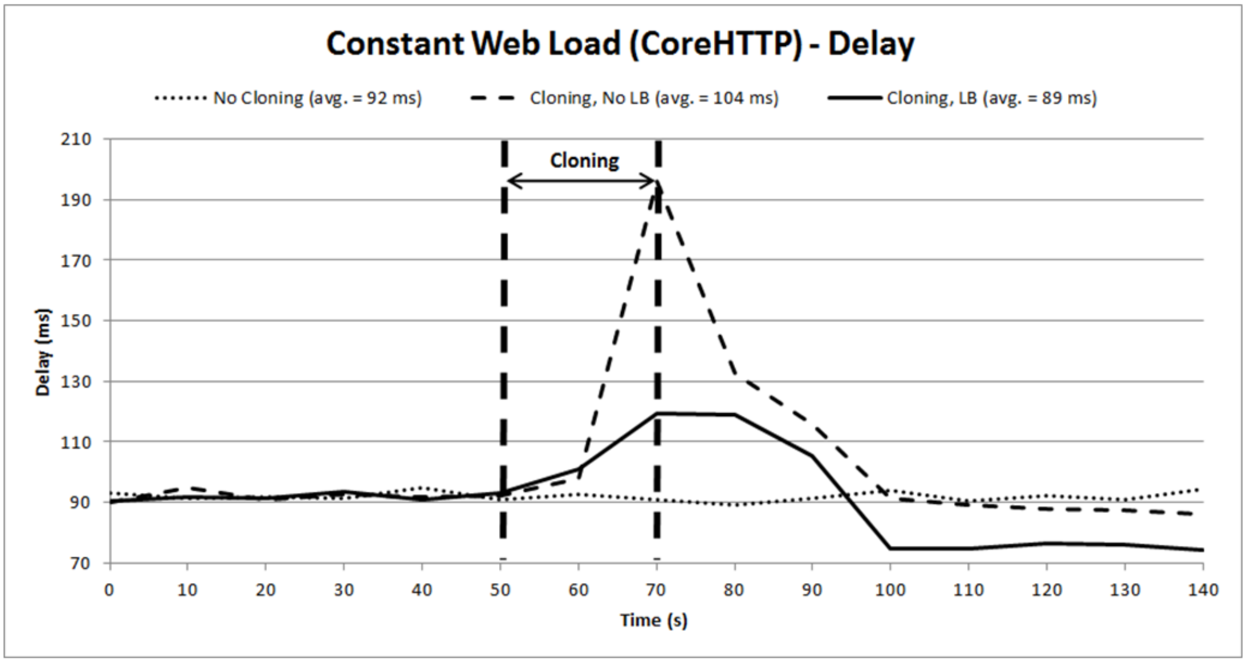


Figure 3.16: GlassFish Server delay under constant web load.

Ever-Increasing Web Load

We now investigate what happens when the web load increases at a constant rate in time. This will give us a sense of the application's maximum capacity under the same three scenarios that were previously tested with a constant web load: (1) no cloning, (2) cloning with no load balancing, and (3) cloning with load balancing.

For these experiments, we expose VMs running GlassFish server to an ever-increasing workload generated using Autobench [2]. Web load on the servers increases at a rate of 1 req/sec. Unlike all previous experiments, these ones use smaller VM instances with 512 MB of RAM, as opposed to the usual 2 GB, in order to allow for a large number of clones to be alive at the same time.

The throughput results are shown in Figure 3.17. From them, we see that the VM Cloning operation, which begins at t=70s, only takes 10 seconds because we are dealing with smaller VM instances than in previous occasions. By the 90-second mark, both scenarios involving cloning have recovered from the cloning procedure's performance impact. At approximately t=110s, a single VM's saturation point of 117 reqs/sec is reached. Without any cloning or the presence of a load balancer, the application cannot keep up with any subsequent increases in the web load, as illustrated by the *No Cloning* and *Cloning, No LB* curves. However, the load-balanced VMs (i.e., *Cloning, LB*) do manage to exceed that threshold and continue increasing their throughput to match the ever-increasing load, being able to keep up with 200 reqs/sec (71% more than a single VM's saturation point) without showing any signs of saturation.

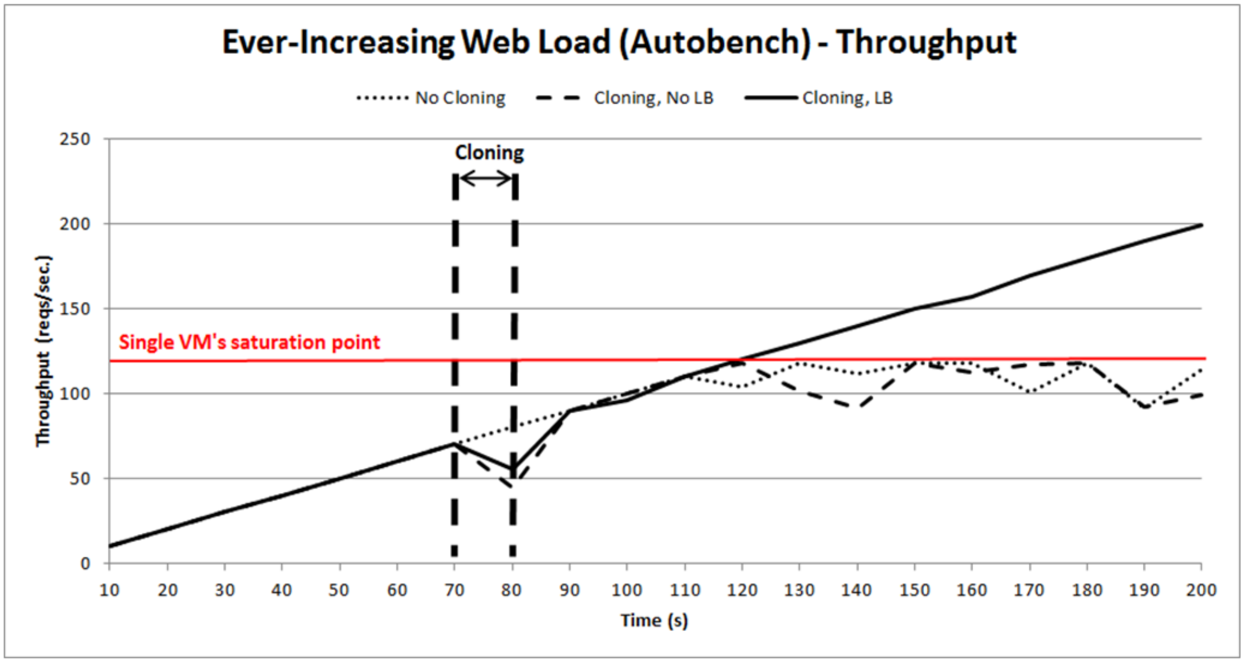


Figure 3.17: GlassFish Server throughput under ever-increasing web load.

Delay measurements for the same experiments are shown in Figure 3.18. As expected, the VM Cloning operation leads to a transient spike in the application's delays. At the beginning, delays are consistently below 10 ms, but the cloning procedure at t=70s generates peak delays in the order of 400 ms that gradually fade away until they are back in the close-to-zero zone just 20 seconds after cloning was triggered. What is more interesting is what happens when a single VM's saturation point of 117 reqs/sec is reached. Given that the *No Cloning* and *Cloning, No LB* arrangements cannot handle any more load from that point onwards, all subsequent load increases translate into higher delays. These new delays are the result of ever-growing packet queues on the server's side. In contrast, the load-balanced cloning setup gracefully handles loads beyond the single VM's saturation point and maintains good quality of service, with delays remaining below the 10 ms mark.

To further test the *micro-elasticity* offered by Pwnetizer's VM Cloning strategy, we decided to spawn a new load-balanced clone VM every 30 seconds until we reached the load balancer's saturation point. Figure 3.19 shows the experiment's outcome. The continuous line corresponds to the application's throughput, while the dotted line illustrates the application's delay. As can be

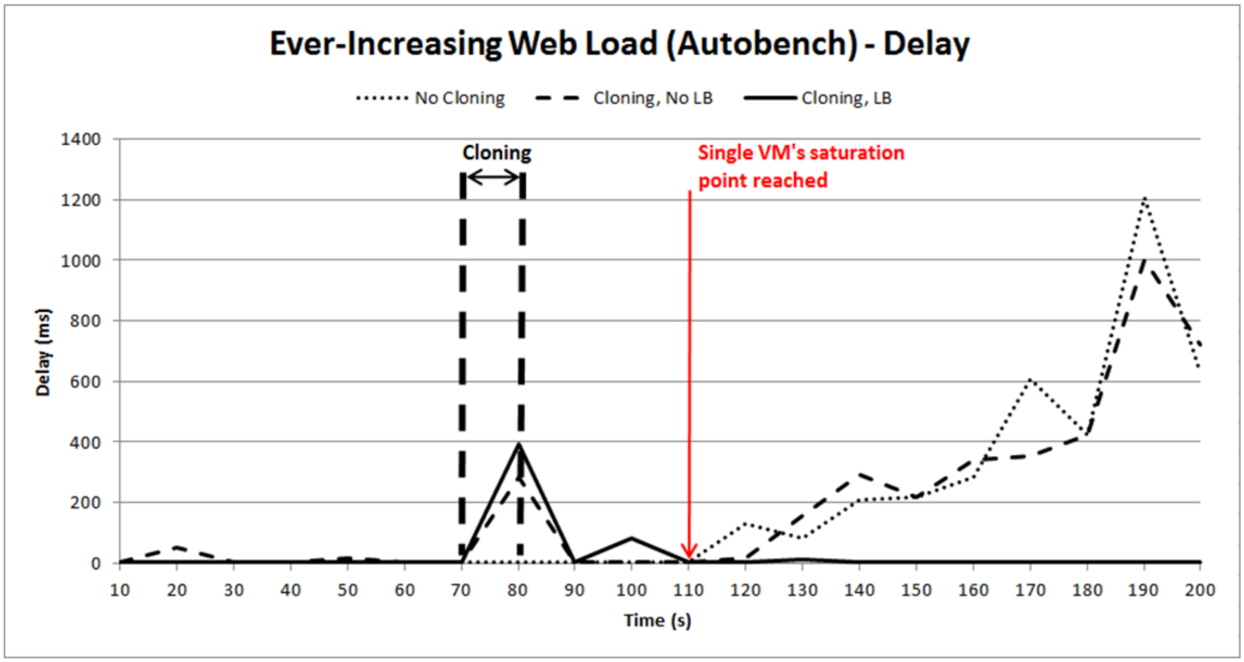


Figure 3.18: GlassFish Server delay under ever-increasing web load.

seen, our cloning mechanism allows us to scale cloud services in real time, providing external clients with the illusion of them interacting with a single large-capacity VM. Without cloning, the static nature of a VM's allocated resources is what limits a web application's performance. With cloning, the cloud provider's network becomes the bottleneck, which is an easier problem to fix.

Computational Workload

For our last cloning experiment, we look at a parallelizable computationally-intensive workload. With only 183 lines of Java code and leveraging the Cajo Project [7] library for distributed computing, we developed an application capable of calculating SHA-256 hashes on large inputs in a master-worker arrangement. New *worker nodes* announce themselves using multicast packets and the *master node* dynamically distributes the computational load across all active workers. This is similar to what is done by other Big Data processing frameworks, such as MapReduce [13], Piccolo [51], CIEL [44] and Spark [70].

Figure 3.20 illustrates a standard SHA-256 workload's progress with and without cloning. The continuous line corresponds to a trial in which the creation of a worker VM's clone starts at t=0, while the dashed line is the result obtained when a single worker VM carries out the entire computation without any interruptions. From the graph, it can be seen that the clone VM comes alive after approximately 12 seconds, as evidenced by a change in the *Clone at 0 sec* line's slope. From then

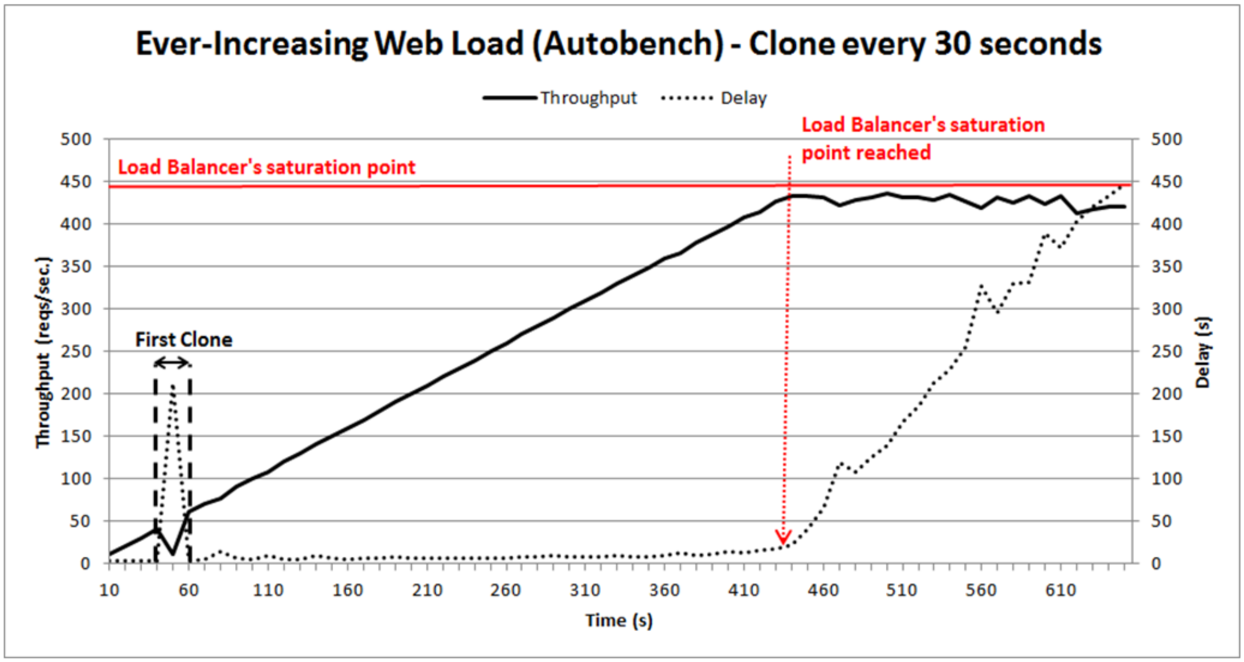


Figure 3.19: GlassFish Server throughput and delay under ever-increasing web load with cloning taking place every 30 seconds.

onwards, the clone VM allows for a faster overall throughput than can be accomplished with a single worker VM, which translates into a 15% improvement in the workload's final running time. Thus, on top of absorbing the transient performance overhead of the cloning procedure, we are also able to reap considerable benefits from the additional VM in under 60 seconds. It should be noted that the performance gain is even larger when running a longer workload.

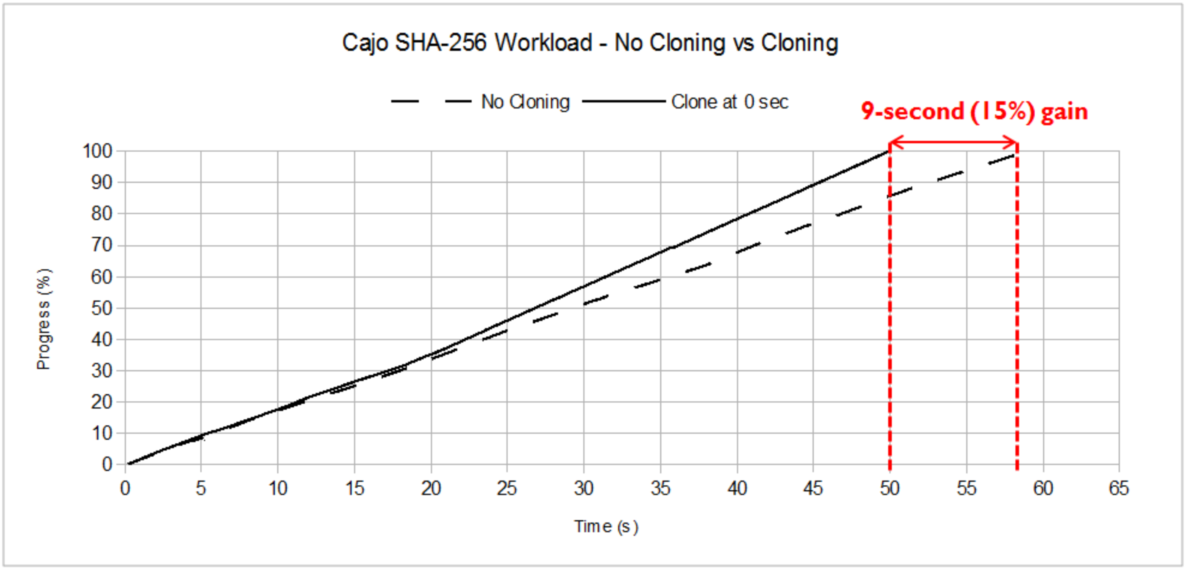


Figure 3.20: Cajo SHA-256 workload progress with and without cloning.

3.9 Related Work

Recent work on VM Cloning has concentrated on parent-child and master-worker arrangements. SnowFlock [34] was the first project to define the VM Forking primitive. The cloning operation is initiated by an application running inside the VM using the *fork API.* Clone VMs are created by way of a modified post-copy live migration procedure, where a minimal processor state is transferred to the remote host and the clone VM is immediately resumed. Each of the clone VM's memory pages must be fetched over the network from its parent VM the first time they are accessed, so the cloning procedure's completion time is unbounded unless page prefetching strategies are employed. SnowFlock requires a custom guest OS and modified application code, along with a highly-customized Xen to work. Consequently, the cloud user's choice of OS is limited and the cloud provider has to stick with an old version of Xen, which may raise security concerns.

Kaleidoscope [5] clones VMs into fractional workers that are single-purpose and transient in order to handle load spikes. It uses SnowFlock, but introduces *VM state coloring*, a technique that bridges the gap between Hypervisor and OS knowledge. By reading metadata in the x86 page table entries, they can categorize pages into 5 *colors*: Kernel Code, User Code, Kernel Data, User Data, and Page Cache Data. That way, they do not need to copy the entire parent VM's memory, but only the subset that is required for the worker to fulfill its task. Once the worker is done, it is powered off. For every parent VM (i.e., booted from scratch), you have N worker clones and a gateway VM. The gateway VM distributes the load between the workers and is the interface with the outside world. The gateway VM and parent VM act as single points of failure, which is a disadvantage from a security perspective. The clone VMs are not self-sufficient and will not function without access to the parent VM's memory pages.

Flurry DB [42] leverages SnowFlock to dynamically scale a MySQL database while presenting a consistent view of the system to external clients. It uses a cloning-aware proxy that issues read operations to any one of the clone VMs, but redirects write requests to all the clones. FlurryDB is a good example of how a complex stateful service can be modified to take advantage of VM Cloning for increased throughput.

Other research has focused on efficiently creating clone VMs on top of the same host. Sun *et al.* [59] and the Potemkin project [64] both take advantage of copy-on-write memory pages to spawn clones with sub-second downtimes. While the Potemkin project maximizes memory sharing amongst the clone VMs in order to emulate thousands of internet honeypots using only a handful of physical servers, Sun *et al.*'s approach supports more general applications and minimizes the performance impact that cloning has on the parent VM. Although very low cloning times are achieved by these works, all clones of a particular VM instance depend on a single host, which does not provide good *crash resilience* guarantees and limits the extent to which *micro-elasticity* can be achieved because the host's physical resources can easily become a bottleneck.

Researchers have also looked at the creation of fully-independent clones on remote hosts. CloneScale [58] supports two types of VM Cloning on a KVM prototype: hot cloning and cold cloning. The hot cloning variant stops a VM temporarily and transfers the main memory contents to an external state file, which can then be used to power on fully-stateful clones. The hot cloning procedure incurs in downtimes of approximately 35 seconds with a 1GB VM, so it is not very lively. Cold cloning, on the other hand, does not suspend any running VMs; it uses a pre-configured VM template. Thus, cold clones can be launched in a matter of seconds, but they do not resemble the state of any of the active VMs.

The most common use of VM Cloning seen in commercial products is based on passive replication. Similar to what is done by Microsoft's Hyper-V and VMWare's vSphere, REMUS [9] maintains a fully-consistent VM replica on a separate host to enable fast failover in case a security-critical VM goes down. The replica is a full VM clone that remains powered down as long as its parent VM is alive. Every set number of milliseconds, an incremental pre-copy operation is carried out to update the replica with the active VM's most recent main memory state. Unfortunately, REMUS replicas do not contribute to the overall application throughput. Furthermore, keeping them up-to-date is costly, with performance overheads on a VM's internal workloads of 103% when taking 40 checkpoints per second [9].

Chapter 4

Improved Availability in Commodity Hypervisors Through I/O Randomization

In Chapter 2, we identified **Availability** and **I/O and network device emulation** as the two main security weaknesses of commodity Hypervisors (see Sections 2.3 and 2.4.1). Chapter 3 used this as a motivation for the development of Pwnetizer, a new VM Cloning technique that can be used to provide increased *availability* for a cloud customer's applications and data in the form of *crash resilience* and *micro-elasticity*. Now, we take advantage of our new VM Cloning technique and complement it with a randomization-based strategy in order to reduce the impact of *I/O and network device emulation* vulnerabilities.

4.1 I/O & Network Device Emulation

This section gives a short overview of how *I/O and network device emulation* works in commodity Hypervisors. It then points out the reasons why this core Hypervisor functionality constitutes a weak point from a security perspective.

4.1.1 Normal Operation

Given that a VM's OS is usually unaware of the fact that it is running on a virtualized environment, I/O device emulation is necessary in order for the Hypervisor to mediate all VM accesses to the host's physical resources. If we provided a VM with direct access to the host's I/O devices, the VM's actions would interfere with tasks being run by the host and other co-hosted VMs on the same devices. Furthermore, this situation would enable any VM to read/write data or send/receive network packets on behalf of other co-hosted VMs, which would be a serious integrity and confidentiality breach.

Hypervisors enforce access policies and multiplex a host's physical devices through a design based on the division of labor. They make use of two types of drivers: front-end and back-end. Front-end drivers cannot access physical hardware directly; they run inside the guest VMs and provide the usual abstractions that a guest OS expects. Back-end drivers run inside a more privileged runtime space (i.e., Dom0 in Xen and Hypervisor space in KVM) with full access to the host's hardware. They communicate with a VM's front-end drivers, forwarding I/O requests and replies to and from the physical I/O devices. Figure 4.1 highlights the interactions between front-end and back-end drivers in Xen and KVM.

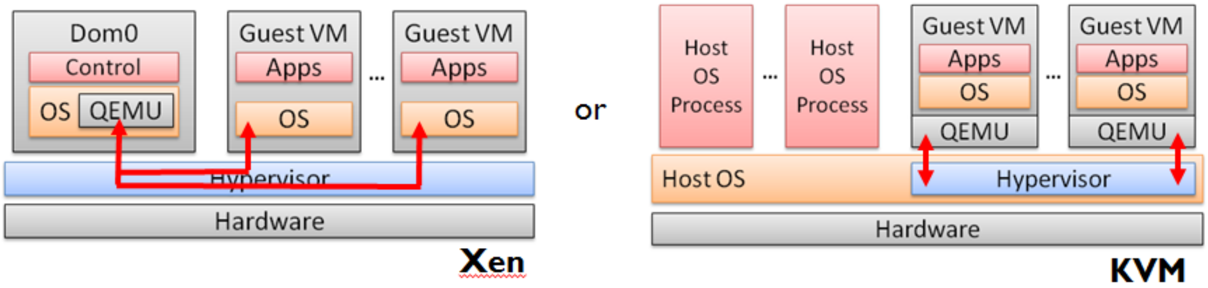


Figure 4.1: Interactions between front-end and back-end drivers in Xen (left) and KVM (right).

4.1.2 Vulnerabilities

In terms of lines of code, the *I/O and network device emulation* module is the largest Hypervisor module in both Xen and KVM; hence, bugs are more likely to occur there. The module's size is explained by the fact that it is in charge of emulating many different types of devices (i.e., network, video, audio, usb, etc). In addition, *I/O & network device emulation* is usually implemented in higher-level languages (e.g., C and C++), which have richer data abstractions and give way to more elaborate attacks. In Section 2.4.1, we found that more than one third of all Hypervisor vulnerabilities studied used *I/O and network device emulation* as their attack vector.

When compromising a Hypervisor's *I/O and network device emulation* functionality, there are two approaches: (1) employing a rogue front-end driver inside a guest VM and (2) attacking the back-end driver from the outside world. For example, a NIC back-end driver must contemplate the possibility of being exposed to malicious front-end drivers as well as network-bound attacks.

4.2 I/O Driver Randomization

A vulnerable back-end driver is the root cause of any breach based on *I/O and network device emulation*. In the case of commodity Hypervisors, there are no provably secure back-end drivers at the user's disposal. However, we find a large number of virtual I/O devices with overlapping functionality (e.g., e1000, ne2k_pci and rtl8139 networking cards), each of them implemented as a unique [front-end, back-end] driver pair. This further increases the device emulation module's size in terms of lines of code, but might be advantageous when clone VMs come into play.

4.2.1 Basic Intuition

Virtual I/O devices are practically independent from the host's physical hardware. As long as the capabilities presented to the VMs by a virtual device are a subset of the host's actual capabilities, the virtual device can be used. Back-end drivers use normal C++ read/write instructions, which in turn make use of the drivers loaded by the host's OS to communicate with the actual hardware. As a result, the hardware's make and model has little influence over the selection of back-end drivers that can be employed. For example, a virtualized Intel Ethernet i82559C network card can be presented to VMs running on top of a host with Realtek RTL8139 Ethernet cards in its physical hardware, assuming that the Intel Ethernet i82559C back-end driver does not expose any advanced features that are not supported by Realtek RTL8139 Ethernet cards. This means that there are many possible combinations of back-end drivers that can be selected by the Hypervisor in order to satisfy the I/O needs of a specific VM on top of a single host. For example, KVM has 8 different NIC back-end drivers: i82551, i82559er, virtio, ne2k_pci, i82557b, rtl8139, e1000, and pcnet. Any one of them can be presented to a VM to provide it with internet connectivity on a typical host with a modern NIC.

Each back-end driver is susceptible to a different set of vulnerabilities. Thus, if we launch every clone VM with a new set of randomly-picked back-end drivers that still satisfy the original VM's needs, we will end up with a diversified group of clones. This way, we can defend against current and future attacks on back-end drivers by spreading out the risk across all clone VMs, which reduces the likelihood of large-scale VM crashes. This defense strategy has the added advantage of not requiring us to vet any of the drivers beforehand.

4.2.2 Actual Implementation

For our proof-of-concept prototype, we opted to concentrate on NIC driver randomization, since network cards are crucial I/O devices in a Cloud Computing scenario. We began by randomly switching the clone VM's back-end driver before powering on the clone VM. This proved to be problematic, given that the VM's OS had already loaded another NIC's front-end driver. Furthermore, standard operating systems detect network cards during boot-up and rule out the possibility of them changing during runtime, so the clone's OS would start complaining about the NIC's voltage being different to what had been measured during boot-up and would then deactivate all networking services.

Our driver randomization solution ended up being simpler. We start every VM instance with several network cards, each card corresponding to a different [front-end, back-end] driver pair. They all carry the exact same MAC address and only one of them is used by the VM at any given time; the rest remain disabled. Inside the VM, a modified PwnetizerClient (see Section 3.5.6) awaits for cloning to take place. If a VM's PwnetizerClient instance detects cloning and concludes that it is dealing with the clone VM, it (1) turns off all network cards, (2) changes all of their MAC addresses to the appropriate one (see Section 3.6), and (3) randomly picks one of the cards as the new active NIC. For this solution to work, we had to increase KVM's hard-coded limit of 8 PCI devices in order to be able to instantiate VMs with all possible NIC models as separate devices. No changes had to be made to the guest OS. Finally, this solution's additional overhead with respect to non-randomized cloning is of only 20 milliseconds, which we consider to be inconsequential.

4.3 Experimental Evaluation

To test the effectiveness of I/O driver randomization in dealing with device emulation attacks, we test our enhanced Pwnetizer prototype against an actual attack. CVE-2010-0741 [41] reported that an improper implementation of TCP Segment Offloading (TSO) in NIC back-end drivers allowed remote attackers to cause a full guest OS crash by sending a large amount of network traffic to a TCP port. This vulnerability affected 3 of KVM's network card back-end drivers (i82551, i82559er, virtio), while the other 5 drivers (ne2k_pci, i82557b, rtl8139, e1000, pcnet) were unaffected.

4.3.1 Methodology

CVE-2010-0741 caused guest VMs running Linux 2.6.27 and older to crash when their virtual NICs were saturated with TCP traffic. Thus, we use Ubuntu 8.04 Hardy LTS (Linux kernel 2.6.24) inside our VMs. During our experiments, we spawn a new clone every 30 seconds and continuously attack all clone VMs. We keep an initial VM with all of its network interfaces disabled, which makes it immune to the attack. That way, the initial VM can be used as the source for every cloning procedure.

4.3.2 Clone Liveliness

We start off with a simple experiment. Every clone VM sends a liveness signal of 90,000 bps to an external server. The final objective is to obtain 5 live VMs at some point in time.

Figure 4.2 contrasts the results of employing randomized clones (continuous line) with the results offered by the non-randomized alternative (dashed line). As can be observed, an original VM with a vulnerable back-end driver always generates clones with the same flaw under the absence of I/O randomization, leading to liveness signal bursts lasting no more than 15 seconds. Consequently, the non-randomized approach offers intermittent uptimes under attack conditions and fails to reach the goal of having 5 live VMs at the same time. On the other hand, our randomized strategy yields more promising results, eventually providing us with 5 clone VMs that are stable regardless of the fact that the attack is still going. Even though the clone VMs that were created at t=33s, t=155s, t=343s, t=371s and t=430s selected vulnerable NIC drivers and ended up crashing, every clone VM that picked an unaffected driver stayed up indefinitely waiting for other ones to join it.

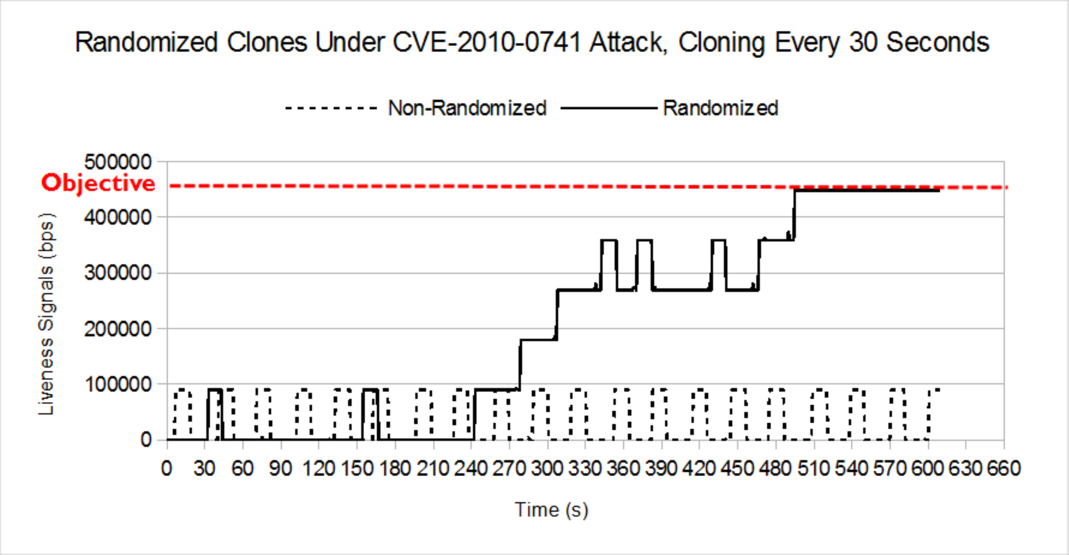


Figure 4.2: Liveliness signals recorded with and without I/O driver randomization. Clones spawned every 30 seconds.

4.3.3 Computational Workload

To analyze the effects of randomized I/O over an actual application's performance, we ran the Cajo SHA-256 workload used in Section 3.8.2 with the randomized and non-randomized cloning approaches. Figure 4.3 illustrates the workload's progress as a function of time. In the *Non-Randomized* curve, many sections with horizontal slope can be identified, which indicate a temporary absence of worker nodes. Meanwhile, the *Randomized* curve shows steady progress starting from t=202s, which can be explained by the presence of a clone VM with an unaffected NIC driver. At the end, the non-randomized clones took 1238 seconds (224%) longer than the randomized clones to complete the same computational task.

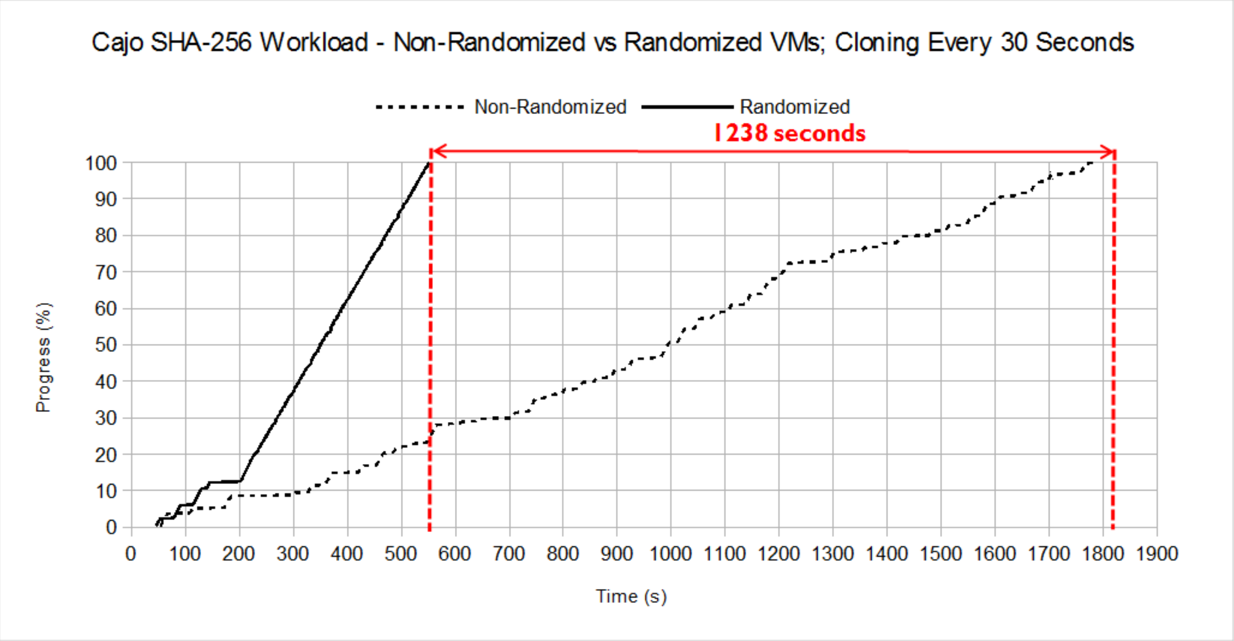


Figure 4.3: Cajo SHA-256 workload progress with and without I/O driver randomization. Clones spawned every 30 seconds.

These experiments prove that our I/O randomization strategy significantly improves an application's ***availability*** *and* ***throughput*** *when* ***I/O and network device emulation attacks*** *are being used against it.*

Chapter 5

Closing Words

5.1 Conclusions

To successfully compromise a system, malicious users must characterize the attack surface available to them and evaluate their possible targets while considering the restrictions of their vantage point (trigger source). In Chapter 2, we conducted a thorough analysis of the codebase of two popular Hypervisors, Xen and KVM, followed by an extensive study of vulnerability reports associated with them. Based on our findings, we are the first to propose and integrate three Hypervisor Vulnerability classifications: by Hypervisor functionality, by trigger source, and by attack target. Our integration of these three classifications gives a clear picture of the different Hypervisor modules and runtime spaces that are traversed during the course of a successful attack. We demonstrated the practicality of this abstract model for vulnerability analysis by describing the flow of events involved in two well-known attacks that achieved privilege escalation in virtualized systems. By clearly exposing potential attack paths, our Hypervisor vulnerabilities characterization is also actionable: it enables us to see what vulnerabilities have been covered by proposed solutions in past work, and what needs to be covered by new defenses. Our work can assist in better establishing a specific user's security needs and determining the scope of the solutions that might be proposed to address them.

Our Hypervisor vulnerability study brought two core security issues in virtualized systems to our attention: *availability* and *I/O & network device emulation*. With *availability* in mind, we developed Pwnetizer, a VM Cloning mechanism that can improve both performance and security. Our cloning technique creates full, independent, and active clone VMs that help us fulfill both of our availability objectives (*crash resilience* and *micro-elasticity*) when proper inter-clone collaboration and data replication mechanisms are in place. Our benchmarks show that we can consistently achieve negligible downtimes with typical Cloud Computing workloads. Total cloning times range between 10 and 33 seconds, which is a considerable improvement from other similar cloning strategies. Lastly, the Pwnetizer prototype has been integrated into a full-blown Cloud Computing environment (i.e., OpenStack) to increase its usability and facilitate future research. Our experiments with real web-based and computationally-intensive applications demonstrate that an application's performance (i.e., throughput, delay, running time) can be significantly improved by leveraging Pwnetizer's on-demand *micro-elasticity*.

By extending the Pwnetizer cloning technique to launch every clone VM with a new set of randomly-chosen back-end drivers, we effectively manage to protect a set of VMs against current and future attacks exploiting *I/O & network device emulation* vulnerabilities. Without having to vet the device drivers beforehand, our I/O randomization extension yields better liveliness and throughput metrics for applications under attack. This fact was proven by running experiments that tested our enhanced Pwnetizer prototype against an actual device emulation attack.

5.2 Future Work

While we have successfully characterized vulnerabilities present in two open-source Hypervisors, we invite closed-source Hypervisor vendors, such as Microsoft and VMWare, to employ our methodology and share their results with the academic community. This would allow us to determine whether or not a high proportion of *availability* and *I/O and network device emulation* threats are a common denominator for virtualized systems as a whole. Furthermore, having access to a closed-source Hypervisor's vulnerability distributions would enable us to discover any important differences between open-source and closed-source virtualization practices.

In terms of Pwnetizer VM Cloning, there are many optimizations that can be made to extend its applicability as a security mechanism. Our development efforts focused on the creation of clones for one VM at a time. We should also consider the scenario where more than one VM is to be cloned from the same Hypervisor, which is a plausible occurrence inside a public or private cloud running multiple security-sensitive VMs. Deshpande *et al.* [15] tackled this problem with respect to Live VM Migration by de-duplicating identical content across VMs, resulting in a 45% improvement in the total migration time for the simultaneous migration of 24 co-hosted 1GB VMs with respect to KVM's default algorithm. Something similar could be done in the VM Cloning domain.

A non-trivial question is: *when should we trigger VM Cloning?* This requires a detailed analysis of different VM workloads and usage scenarios. For example, it would be interesting to employ load-prediction algorithms in order to preemptively trigger cloning before critical levels of network bandwidth, CPU, and/or RAM usage are reached. Furthermore, this sort of analysis would have to be coupled with a characterization of the costs associated with VM Cloning inside a public or private cloud. Given that clone VMs consume cloud resources, their lifespan should be restricted to time intervals in which they are completely indispensable for a cloud customer's applications to provide adequate quality of service.

Last but not least, our I/O randomization prototype focused on NIC driver randomization. However, many other types of devices could benefit from this technique. For instance, USB back-end drivers could be targeted by an attacker with physical access to the servers. Our randomization idea could mitigate the risk of such an insider threat.

Appendix A

Sample CVE Reports

Table A.1 lists details about the CVE reports mentioned in Section 2.4, taken **verbatim** from [46], [55], [52] and [10].

Table A.1: Sample CVEs in Support of the Functionality-Based Classification

Virtual CPUs: CVE-2010-4525 (KVM-related)
Some versions of the Linux kernel forgot to initialize the kvm_vcpu_events.interrupt.pad field before being copied to userspace. kvm_vcpu_events.interrupt.pad field must be initialized before being copied to userspace, otherwise kernel memory is leaked.
SMP: CVE-2010-0419 (KVM-related)
The x86 emulator in KVM 83, when a guest is configured for Symmetric Multiprocessing (SMP), does not properly restrict writing of segment selectors to segment registers, which might allow guest OS users to cause a denial of service (guest OS crash) or gain privileges on the guest OS by leveraging access to a (1) IO port or (2) MMIO region, and replacing an instruction in between emulator entry and instruction fetch.
Soft MMU: CVE-2010-0298 (KVM-related)
Gleb Natapov found a bug in KVM that allows code that runs in CPL3 (inside a guest) to modify memory in CPL0 (inside a guest). The bug is in x86 emulator code. When emulator accesses guest's memory on behalf of the guest's code it does this with the CPL0 privilege, so if emulated instruction is executed by unprivileged code it can still modify memory that otherwise is not accessible to it.
Interrupts and Timers: CVE-2010-0309 (KVM-related)
A flaw was found in the Programmable Interval Timer (PIT) emulation. Access to the internal data structure pit_state, which represents the data state of the emulated PIT, was not properly validated in the pit_ioport_read() function. A privileged guest user could use this flaw to crash the host.
I/O and Networking: CVE-2011-1751 (KVM-related)
Writing the value 2 to I/O port 0xae08 initiates the PIIX4 PCI-ISA bridge removal. Unplugging this causes all of the ISA devices to be unplugged and right now the ISA (in particularly the RTC) devices cannot handle unplug gracefully.
Paravirtualized I/O: CVE-2008-1943 (Xen-related)
The PVFB backend is a user space program running as root in dom0. A buggy or malicious frontend can describe its shared framebuffer to it in a way that makes it map an arbitrary amount of guest memory, malloc an arbitrarily large internal buffer, or copy arbitrary memory to that buffer.
VM Exits: CVE-2010-2938 (Xen-related)
When an Intel platform without Extended Page Tables (EPT) functionality is used, the virtual-machine control structure (VMCS) implementation accesses VMCS fields without verifying hardware support for these fields, which allows local users to cause a denial of service (host OS crash) by requesting a VMCS dump for a fully virtualized Xen guest.

Hypercalls: CVE-2009-3290 (KVM-related)
The *kvm_emulate_hypercall* function in arch/x86/kvm/x86.c in KVM in the Linux kernel 2.6.25-rc1, and other versions before 2.6.31, when running on x86 systems, does not prevent access to MMU hypercalls from ring ¿ 0, which allows local guest OS users to cause a denial of service (guest kernel crash) and read or write guest kernel memory via unspecified ”random addresses.”
VM Management: CVE-2007-4993 (Xen-related)
Pygrub (tools/pygrub/src/GrubConf.py) in Xen 3.0.3, when booting a guest domain, allows local users with elevated privileges in the guest domain to execute arbitrary commands in domain 0 via a crafted grub.conf file whose contents are used in exec statements.
Remote Management SW: CVE-2008-3253 (Xen-related)
Cross-site scripting (XSS) vulnerability in the XenAPI HTTP interfaces in Citrix XenServer Express, Standard, and Enterprise Edition 4.1.0; Citrix XenServer Dell Edition (Express and Enterprise) 4.1.0; and HP integrated Citrix XenServer (Select and Enterprise) 4.1.0 allows remote attackers to inject arbitrary web script or HTML via unspecified vectors.
Hypervisor Add-Ons: CVE-2008-3687 (Xen-related)
Heap-based buffer overflow in the flask_security_label function in Xen 3.3, when compiled with the XSM:FLASK module, allows unprivileged domain users (domU) to execute arbitrary code via the flask_op hypercall.

Appendix B

The Different Stages of VM Cloning

Below is a short description of each Pwnetizer OpenStack stage marked in Figure 3.14:

1. **TRIGGER DELAY 1** – Time taken for the *source Nova-Compute*'s message to go through OpenStack's messaging queue and reach the *target Nova-Compute*.
2. **TRIGGER DELAY 2** – Time taken for the *source Nova-Compute* to communicate with PwnetizerLibvirt and trigger the cloning operations.
3. **SETUP TIME 1** – Interval during which the original VM's XML description is generated by the source Hypervisor and interpreted by the target Hypervisor on the other host.
4. **DB OVERHEAD 1** – PwnetizerServer queries Nova DB to find the next available *VM id*, which is used to generate an OpenStack-compliant name for the clone VM.
5. **CLONE DEFINITION AND ANNOUNCEMENT** –The target Hypervisor sends PwnetizerServer all the information required for it to carry out the appropriate *disk cloning* tasks.
6. **STORAGE INITIALIZATION** – PwnetizerServer moves copies of the original VM's disk files from the local mirror into the NFS share, so that they can be used by the clone VM. Appropriate storage synchronization threads are spawned.
7. **CLONE LAUNCH** – The clone VM is launched in suspended state on the target Hypervisor.
8. **MAIN PRE-COPY** – Main memory pages are iteratively sent to the target Hypervisor using the pre-copy algorithm while the original VM remains powered on.

9. **FINAL PRE-COPY** – The source Hypervisor pauses the original VM and performs the last pre-copy iteration. This is where the downtime begins.

10. **STORAGE FINALIZATION** – Final synchronization calls are issued to ensure full consistency between the original VM's and the clone VM's disk files.

11. **RESUME** – The original VM is resumed, putting an end to the cloning procedure's downtime.

12. **DB OVERHEAD 2** – PwnetizerServer updates the original VM's state in Nova DB as a way of signaling the *source Nova-Compute* to wake up. The clone VM's information is also added to Nova DB.

13. **DETECTION OVERHEAD** – Time taken for the *source Nova-Compute* to notice that the original VM is active again.

14. **DB OVERHEAD 3** – Minor Nova DB update issued by the *target Nova-Compute* so that the clone VM's information is complete.

15. **NOVA-NETWORK** – The *target Nova-Compute* asks for all the network resources required by the clone VM and waits for Nova-Network to fulfill the request.

16. **DB OVERHEAD 4** – The *target Nova-Compute* queries Nova DB to find the MAC address that the clone VM must acquire in order to obtain a valid DHCP lease. It then sends a *broadcast* packet with sufficient information for the clone VM and original VM to realize that cloning has taken place and take the appropriate actions.

17. **NETWORK RECONFIGURATION** – Time taken by the clone VM to change its MAC address and obtain a new network identity.

Bibliography

[1] The Apache HTTP Server Project. http://httpd.apache.org/.

[2] Autobench. http://www.xenoclast.org/autobench/.

[3] 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 ACM Conference on Computer and Communications Security*, CCS, pages 38 – 49, October 2010.

[4] Rohit Bhadauria, Rituparna Chaki, Nabendu Chaki, and Sugata Sanyal. A survey on security issues in cloud computing. *arXiv, http://arxiv.org/abs/1109.5388*, September 2011.

[5] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 273–286, New York, NY, USA, 2011. ACM.

[6] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[7] Cajo, the easiest way to accomplish distributed computing in Java. http://www.javacodegeeks.com/2011/01/cajo-easiest-way-to-accomplish.html.

[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 & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[9] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[10] Cve security vulnerability database. http://www.cvedetails.com/.

[11] W. Dawoud, I. Takouna, and C. Meinel. Infrastructure as a service security: Challenges and solutions. In *Proceedings of the International Conference on Informatics and Systems*, INFOS, pages 1 – 8, March 2010.

[12] Dbench filesystem benchmark. http://dbench.samba.org/.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[15] Umesh Deshpande, Xiaoshuang Wang, and Kartik Gopalan. Live gang migration of virtual machines. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 135–146, New York, NY, USA, 2011. ACM.

[16] Dnsmasq - a DNS forwarder for NAT firewalls. `http://www.thekelleys.org.uk/dnsmasq/doc.html`.

[17] N. Elhage. Virtunoid: Breaking out of KVM. `nelhage.com/talks/kvm-defcon-2011.pdf`, August 2011.

[18] Faban Harness and Benchmark Framework. `http://java.net/projects/faban/`.

[19] P Ferrie. Attacks on virtual machine emulators, white paper, symantec corporation, january 2007.

[20] Peter Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[21] D.E. Geer. Attack surface inflation. *IEEE Security Privacy Magazine*, 9(4):85 – 86, July – August 2011.

[22] GlassFish - Open Source Application Server. `http://glassfish.java.net/`.

[23] N. Gruschka and M. Jensen. Attack surfaces: A taxonomy for attacks on cloud services. In *Proceedings of the IEEE International Conference on Cloud Computing*, CLOUD, pages 276 – 279, July 2010.

[24] HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. `http://haproxy.1wt.eu/`.

[25] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 51–60, New York, NY, USA, 2009. ACM.

[26] Martin Hingley. The OVA pushes KVM as the next big virtualisation ecosystem. `http://rainmakerfiles.com/2012/02/ova/`.

[27] Dawei Huang, Deshi Ye, Qinming He, Jianhai Chen, and Kejiang Ye. Virt-lm: a benchmark for live migration of virtual machine. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, pages 307–316, New York, NY, USA, 2011. ACM.

[28] Nexenta Hypervisor Survey. `http://www.nexenta.com/corp/nexenta-hypervisor-survey`.

[29] Is the Hypervisor Market Expanding or Contracting? `http://www.aberdeen.com/Aberdeen-Library/8157/AI-hypervisor-server-virtualization.aspx`.

[30] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual., October 2011. `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[31] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the Annual International Symposium on Computer Architecture*, ISCA, pages 350 – 361, June 2010.

[32] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 7:1–7:14, Berkeley, CA, USA, 2007. USENIX Association.

[33] Will KVM Follow Hyper-V in Its Adoption Curve? `http://www.clabbyanalytics.com/uploads/KVM_Final.pdf`.

[34] H. Andrés Lagar-Cavilla, Joseph A. Whitney, Roy Bryant, Philip Patchin, Michael Brudno, Eyal de Lara, Stephen M. Rumble, M. Satyanarayanan, and Adin Scannell. Snowflock: Virtual machine cloning as a first-class cloud primitive. *ACM Trans. Comput. Syst.*, 29(1):2:1–2:45, February 2011.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[36] J.F. Levine, J.B. Grizzard, and H.L. Owen. Detecting and categorizing kernel-level rootkits to aid future detection. *IEEE Security Privacy Magazine*, 4(1):24 – 32, January – February 2006.

[37] Chunxiao Li, Anand Raghunathan, and Niraj K. Jha. Secure Virtual Machine Execution under an Untrusted Management OS. In *Proceedings of the Conference on Cloud Computing*, CLOUD, pages 172 – 179, July 2010.

[38] Libvirt: The virtualization API. `http://libvirt.org/`.

[39] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the harp file system. *SIGOPS Oper. Syst. Rev.*, 25(5):226–238, September 1991.

[40] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[41] Petr Matousek. Cve-2010-0741 qemu: Improper handling of erroneous data provided by linux virtio-net driver. `bugzilla.redhat.com/show_bug.cgi?id=577218`.

[42] Michael J. Mior and Eyal de Lara. Flurrydb: a dynamically scalable relational database with virtual machine cloning. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 1:1–1:9, New York, NY, USA, 2011. ACM.

[43] mstone multi-protocol testing system. `http://sourceforge.net/projects/mstone/`.

[44] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[45] MySQL Cluster CGE. `http://www.mysql.com/products/cluster/`.

[46] National vulnerability database. `http://web.nvd.nist.gov/view/vuln/search`.

[47] OpenStack: Open Source Cloud Computing Software. `http://www.openstack.org/`.

[48] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: an efficient and portable web server. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 15–15, Berkeley, CA, USA, 1999. USENIX Association.

[49] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[50] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the USENIX Security Symposium*, pages 179 – 194, August 2004.

[51] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[52] Red hat bugzilla. https://bugzilla.redhat.com/.

[53] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, pages 199 – 212, November 2009.

[54] Joanna Rutkowska and Rafa Wojtczuk. Preventing and detecting xen hypervisor subversions. invisiblethingslab.com/resources/bh08/part2-full.pdf, July 2008.

[55] Securityfocus. http://www.securityfocus.com/.

[56] smtp-sink(1) - Linux man page. http://linux.die.net/man/1/smtp-sink.

[57] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1 – 11, 2011.

[58] Sethuraman Subbiah. Clonescale: Distributed resource scaling for virtualized cloud systems. Master's thesis, North Carolina State University, 2011.

[59] Yifeng Sun, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Binbin Zhang, Haogang Chen, and Xiaoming Li. Fast live cloning of virtual machine based on xen. In *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, pages 392–399, 2009.

[60] SysBench: a system performance benchmark. http://sysbench.sourceforge.net/.

[61] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the Conference on Computer and Communications Security*, CCS, October 2011.

[62] Luis Vaquero, Luis Rodero-Merino, and Daniel Morn. Locking the sky: a survey on iaas cloud security. *Computing*, 91:93 – 118, 2011.

[63] VLC media player. http://www.videolan.org.

[64] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 148–162, New York, NY, USA, 2005. ACM.

[65] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[66] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 158 – 177. 2010.

[67] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*, S&P, pages 380 – 395, May 2010.

[68] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[69] Wireshark: the world's foremost network protocol analyzer. http://www.wireshark.org/.

[70] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[71] Minqi Zhou, Rong Zhang, Wei Xie, Weining Qian, and Aoying Zhou. Security and privacy in cloud computing: A survey. In *Proceedings of the International Conference on Semantics Knowledge and Grid*, SKG, pages 105 –112, November 2010.