Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring

Gary Wang Zachary J. Estrada Cuong Pham Zbigniew Kalbarczyk
Ravishankar K. Iyer
University of Illinois at Urbana-Champaign

Abstract

Security requirements in the cloud have led to the development of new monitoring techniques that can be broadly categorized as virtual machine introspection (VMI) techniques. VMI monitoring aims to provide high-fidelity monitoring while keeping the monitor secure by leveraging the isolation provided by virtualization. This work shows that not all hypervisor activity is hidden from the guest virtual machine (VM), and the guest VM can detect when the hypervisor performs an action on the guest VM, such as a VMI monitoring check. We call this technique *hypervisor introspection* and demonstrate how a malicious insider could utilize this technique to evade a passive VMI system.

1 Introduction

Despite cloud computing's widespread adoption, security in the cloud remains a major concern for companies and organizations. Because public cloud infrastructure may be shared by competing companies, it is not surprising that many organizations are wary of moving data into public clouds. Due to these concerns, public cloud operators are expected to employ various forms of security monitoring to detect malicious activity. To accomplish this, traditional network and host-based Intrusion Detection Systems (IDS) are often deployed. In addition to these monitoring systems, virtualized environments can leverage a unique monitoring technique known as Virtual Machine Introspection (VMI) [3].

In VMI, the security monitor is located at the hypervisor level, and inspects a guest virtual machine (VM). The benefits of VMI are twofold. First, the monitor is isolated from the guest VM, so it is difficult (if not impossible) for a malicious VM to compromise the VMI monitor unless it utilizes a VM escape exploit [5]. Secondly, the VMI monitor has access to the underlying hardware state of the VM because the hypervisor manages the hardware resources allocated to the VM and is responsible for virtualizing any hardware devices. With access to the hardware state of the VM, the monitor can obtain information about the VM's activities, such as current running processes or opened files. This combination of monitor security and fidelity makes VMI an attractive monitoring technique.

There are two kinds of VMI monitoring: active and passive. Active VMI revolves around monitoring checks triggered by certain events (usually in hardware), such as specific hardware registers or memory regions being accessed. Passive VMI, on the other hand, performs a monitoring check on a predefined interval, such as checking what processes are running every second.

Our work demonstrates that not all hypervisor activity is isolated from the guest VM. In particular, we show that it is possible for a guest VM to determine the presence of a passive VMI system and its monitoring frequency through a timing side-channel. We call this technique *hypervisor introspection* (HI), which is the converse of VMI. In addition to HI, we also present two insider attack scenarios that leverage HI to evade a passive VMI monitoring system. Lastly, we discuss current state-of-the-art defenses against side-channel attacks in cloud environments and their shortcomings against HI.

2 Related Work

Previous work looking at side-channel attacks in cloud environments have focused on obtaining information from co-resident VMs via cache-based side-channels. For Infrastructure-as-a-Service (IaaS) clouds, Ristenpart et al. looked at various side-channel attacks to determine co-residency of Amazon EC2 instances [10], and Zhang et al. explored the use of a cache-based side-channel to steal cryptographic keys from a co-resident VM [14]. For newer Platform-as-a-Service (PaaS) clouds, Zhang et al. have looked at extracting secrets and compromising pseudorandom number generators of co-located PaaS tenants [15].

Other previous work has also looked at determining the existence of a hypervisor (i.e., determining whether or not an environment is virtualized). One can determine the presence of a hypervisor by running code that will induce hypervisor overhead, and timing the execution of such code [2].

Our work differs from these previous work in that we look to use a side-channel to determine information about the hypervisor's activities instead of a co-resident VM's activity. Our work goes beyond simply determining the existence of a hypervisor, and instead reaches conclusions about the hypervisor's activity. Thus, the presented work is, to the best of our knowledge, the first to infer hypervisor activity from within the guest VM.

3 Hypervisor Introspection

In this section, we discuss the VMI monitor we developed to test HI against, and the side-channel we used to perform HI. The test system used for all of our experiments was a Dell PowerEdge 1950 server with 16GB of memory and an Intel Xeon E5430 processor 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

3.1 VMI Monitor

In order to test the effectiveness of HI, we implemented a VMI monitor. To accomplish this, we used LibVMI [7]. LibVMI is a software library that helps with the development of VMI monitors. It focuses on abstracting the process of accessing a guest VM's volatile memory. The volatile memory of a VM contains information about the guest VM's operating system (OS), such as kernel data structures, which can be examined to determine runtime details such as running processes or active kernel modules.

Our VMI monitor is built on top of the process listing example included with LibVMI version 0.12. In addition to listing the current running processes of the monitored VM, this VMI monitor lists the number of open sockets (both Unix and TCP) associated with each process. The monitor maintains a whitelist of processes that can have sockets open, and raises an alarm when a non-whitelisted process has an open socket.

This monitor could be used to detect unauthorized processes accessing the network. Network access is typically closely monitored, so the information obtained by our monitor is useful in many scenarios (e.g., security monitoring and auditing). LibVMI is a popular VMI library, so the performance of this monitor is representative of environments utilizing VMI.

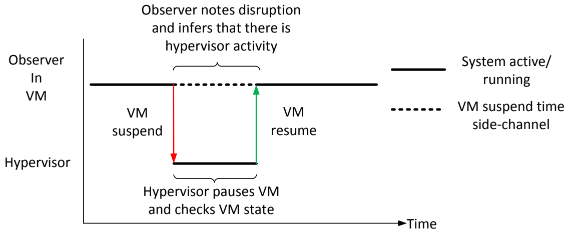
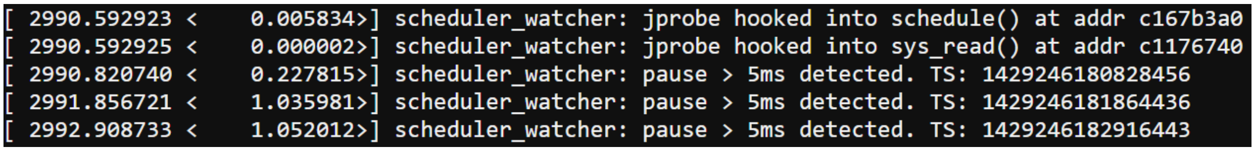

Observer notes disruption and infers that there is hypervisor activity
Observer In VM
VM suspend
VM resume
Hypervisor
Hypervisor pauses VM and checks VM state
Time
System active/ running
VM suspend time side-channel

Figure 1: Illustration of the VM suspend side-channel where an observer notes disruptions in VM activity to determine when the hypervisor is preforming some action on the VM.

Because the polling rate of the VMI monitor is directly related to the performance overhead of the monitor, the polling rate must be chosen carefully so as to not introduce an unacceptable decrease in VM performance. Based on the benchmarking results from [11], there is a large jump in performance overhead (from 5% to 35%) after the polling frequency goes over once per second. Thus, we configured the VMI monitor to poll the guest VM every second. This rate introduced only a 5% overhead in VM performance based on benchmark results from UnixBench [1], which matches the results from [11]. Additionally, this overhead is similar to overhead introduced by an active VMI monitoring system [9].

3.2 VM Suspend Side-channel

Because HI revolves around making measurements from a side-channel and inferring hypervisor activity, we first had to identify the actual side-channel to be exploited. We noted that whenever the hypervisor wants to perform a monitoring check on a guest VM, the VM has to be paused in order to obtain a consistent view of the hardware state. If an observer can detect these VM suspends, then that observer might be able to reach some conclusions about the monitoring checks performed by the hypervisor. We call this side-channel the *VM suspend side-channel*. We came up with two potential methods of measuring the VM suspends: network-based timing measurements and host-based, in-VM timing measurements. Because the network is a noisy medium due to dropped packets and routing inconsistencies, we focused on performing in-VM measurements to detect VM suspends.



```
[ 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 2: Output from the `dmesg -d` command being run after our in-VM measurement kernel module is inserted. 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.

3.2.1 In-VM Timing Measurements

We note that an observer would require access to the VM he would want to perform measurements on, which would be possible in an insider threat or stolen credentials scenario. If the observer would like to perform the measurements without access to the target VM, he could attempt to obtain a co-resident VM [10] and perform the measurements on that VM. Because co-resident VMs run on top of the same hypervisor, the measurements performed on one VM should hold for other co-resident VMs unless the hypervisor is selective in choosing which VMs to monitor.

By timing frequently recurring OS events (e.g., those related to scheduling), the observer can determine when VM suspends occur and infer hypervisor activity. This is illustrated in Figure 1. We chose two events to measure: process scheduling and I/O read operations. These events were chosen since modern OSes run many processes concurrently. These processes need to be constantly scheduled and read data from the disk or other processes. Therfore, both of these events occur at a high frequency during normal OS operation. In our test system, we found that the combined event frequency was around 14300 events per second or one event every 70μs for an idle system.

To measure the frequency of these events, we leveraged *jprobes*. Jprobes are special kernel probes (kprobes) that hook into any kernel function's entry point. When the hooked function is called, execution is redirected to the handler function registered to the jprobe. We created jprobes that hooked into the `schedule` and `sys_read` kernel functions. The jprobe's handler function generated a timestamp using `do_gettimeofday` and checked the timestamp against a shared (i.e., global) "last recorded" timestamp. If the time difference was greater than a certain threshold, then an alert was written out to the kernel log saying a long pause was detected. The last recorded timestamp was then updated regardless of a long pause being detected. We implemented the two jprobes in a kernel module, and example output from these jprobes is shown in Figure 2.

The threshold time was determined empirically against the implemented LibVMI monitor. Our goal was to find a threshold value that was less than the pause introduced by a VM suspend, but greater than the delays between the function calls during normal OS operation (i.e., minimize false positives of suspected VM suspends). In our testing, we found that 5ms was a good threshold value to differentiate VM suspends from delays due to normal OS operation.

3.3 Limitations of Hypervisor Introspection

There are two limitations to HI: the accuracy of the monitoring intervals measured by HI and determining the threshold value for identifying VM suspends. Our testing of HI showed that it is capable of determining monitoring intervals down to 0.1s. This was determined by testing HI against increasingly frequent monitoring checks, and the maximum frequency resolved by HI was one check every 0.1s (10 Hz). The resolution limit of the HI measurements is also present in Figure 2 on the last two lines. The numbers in the angle brackets (i.e., the monitoring interval) are accurate to a tenth of a second.

Because the threshold value for HI is found through empirical testing, it is not straightforward to develop HI for any given system. However, there is some leniency in finding a suitable threshold value. Our testing of HI indicated that a threshold value from 5ms to 32ms yielded the same accuracy when performing HI and detecting the VM suspends. As future work, the threshold value may be correlated with various system specifications, such as kernel version, CPU model/frequency, and system load. After finding threshold values on various systems via empirical testing, a formal relationship between the various system specifications and the threshold value may be derived to obtain a threshold without testing.

4 Evading VMI with Hypervisor Introspection

Although hypervisor introspection can be used to determine the existence of a passive VMI monitoring and its monitoring interval, it may not be clear how this information could be used to actually evade the monitoring system. This section discusses two applications of HI to hide malicious activity from a passive VMI system.

4.1 Insider Attack Model and Assumptions

We present an insider threat attack model where the insider already has administrator (i.e., root) access to VMs running in a company's public IaaS cloud. The insider knows that he will be leaving the company soon, but would like to maintain a presence on the VMs he has access to. The insider does not have access to the underlying hypervisor hosting the VMs, but knows that the company is utilizing some form of passive VMI monitoring. We also assume that the company's VMI monitor is similar to the monitor we implemented, which regularly checks for unauthorized processes running with an open socket. As insiders have full control over their VMs, the company relies on VMI for monitoring, so changes to the VM, such as kernel modifications or new files created, are not detected.

In this attack model, the insider can utilize HI to detect invocations of the monitor and to hide malicious activities from the VMI monitor.

4.2 Large File Transfer

Attackers commonly want to exfiltrate data out of the network after compromising a system. The attacker may leverage various tools and protocols to accomplish this, such as secure copy (SCP), file transfer protocol (FTP), the attacker's own custom utility, or any number of other file transfer methods. A passive VMI system may detect attempts at data exfiltration by maintaining a restrictive whitelist of processes that are allowed to run with a network socket open. A restricted VM would have a whitelist with zero or few entries.

For sufficiently large files being transferred by a non-whitelisted process, the transfer time would be longer than the monitoring interval and the VMI monitor would detect the file transfer. This scenario is illustrated in Figure 3a. We tested the effectiveness of our VMI monitor for this scenario by transferring a 250MB file over TCP from the monitored VM to a co-resident VM, and our VMI monitor detected the Python process running with an open socket and signaled an alarm.

A large file can be transferred over the network without triggering any VMI alarms by splitting the file into chunks and transferring each chunk between monitoring checks. This process is illustrated in Figure 3b. Thus, HI can be used to time the transfer of each chunk to evade the VMI system.

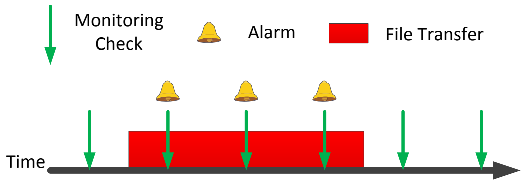


Figure 3a: Illustration of an unauthorized file transfer being detected by the VMI system

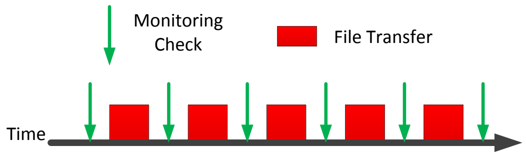


Figure 3b: Illustration of how a large file can be transferred in chunks by a non-whitelisted process without triggering the passive VMI monitor.

We implemented this attack by writing a Python script that transfers a chunk of a file each time it is executed. The offset of the current chunk is written to an "offset log" file after each execution, so that the next execution knows which chunk to transfer. We extend our HI kernel module to call the Python script with the `call_usermodehelper` kernel function after a suspected VM suspend. With our modified HI kernel module and Python script, we were able to transfer the same 250MB file without being detected by our VMI monitor.

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 100 tests) 4.905s with a standard deviation of 0.414s. Using our technique with 8MB chunks, the entire transfer takes 32 calls to the Python script, and thus, 32 monitoring intervals. Because each monitoring interval is 1s long, the entire transfer takes around 32s, which is about eight times the transfer time of the naïve method.

We note that the transfer speed of the large file using this method depends on the polling frequency of the VMI monitor. Lower polling frequencies lead to longer transfer times because each chunk transfer is triggered by a monitoring check, so chunks would be transferred less frequently. However, the attacker can increase the chunk size to counteract this delay after learning the polling frequency through HI. Increasing the chunk size is risky, and it may end up triggering the VMI system, so the attacker would likely verify a larger chunk size on his own system before performing the attack. Figure 4 shows the tradeoffs in increasing the chunk size for the file transfer.

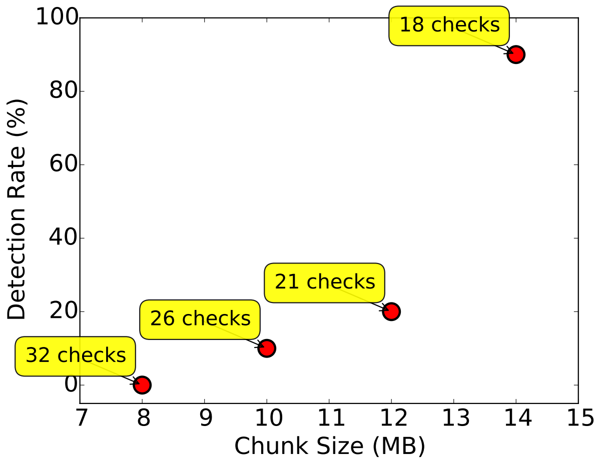


Figure 4: Chunk size versus detection rate for hiding a large file transfer using HI. Note the tradeoff between number of monitoring checks (i.e., transfer time) and detection rate.

Increased chunk size leads to higher detection rate, but the whole file is transferred in fewer monitoring checks.

4.3 Backdoor Shell

In addition to exfiltrating data, attackers typically want to maintain access to compromised systems by installing a backdoor. A naïve backdoor that listens for an incoming connection or connects back to the attacker would be detected by the passive VMI system because the backdoor will have a socket open.

A backdoor can evade the passive VMI system, however, by repeatedly connecting back to the attacker between monitoring checks. A backdoor "client" is run on the VM between the monitoring checks which connects to a backdoor "server" on the attacker's machine. The backdoor server maintains a queue of commands to be run on the VM by the backdoor client. The backdoor client performs a *command cycle* for every two monitoring checks. A command cycle is illustrated in Figure 5 and is made up of the following steps:

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 next monitoring check occurs

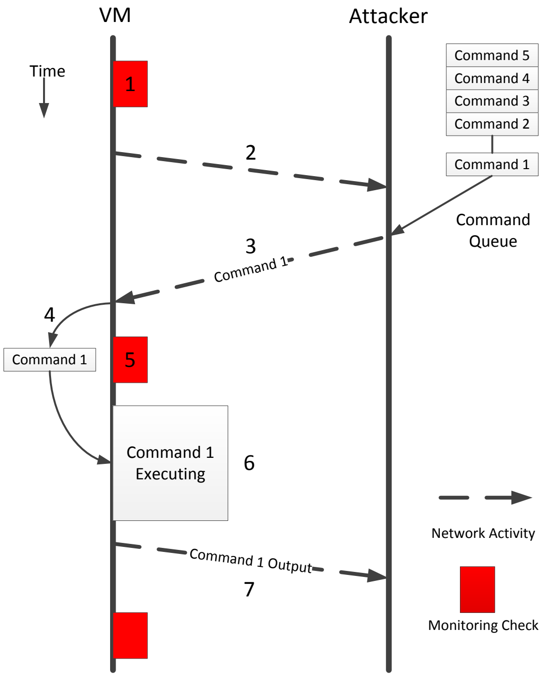


Figure 5: Illustration of the various steps in a command cycle as described in Section 4.3.

6. The backdoor client runs the command it saved

7. The output of the command is sent back to the server

We implemented the backdoor client and server as two Python scripts. The backdoor server maintains a queue for commands that are read from standard input, and the backdoor client either retrieves the next command to be run or executes the current command and sends the output back to the backdoor server. The backdoor client script is called by the HI kernel module with `call_usermodehelper` after each suspected VM suspend.

With this backdoor, the attacker can run commands such as `cat /etc/shadow` and `cat /root/.ssh/id_rsa` to dump password hashes and SSH private keys. While our backdoor client was running, our LibVMI monitor did not raise any alarms across ten tests. Conversely, a naïve backdoor that always listened for incoming connections was detected by the LibVMI monitor in all ten tests.

Although this backdoor lets some commands be run on the VM, longer running commands would still be detected by the VMI monitoring system because the sending of the command output would be too late and occur during the monitoring check. The command cycle could

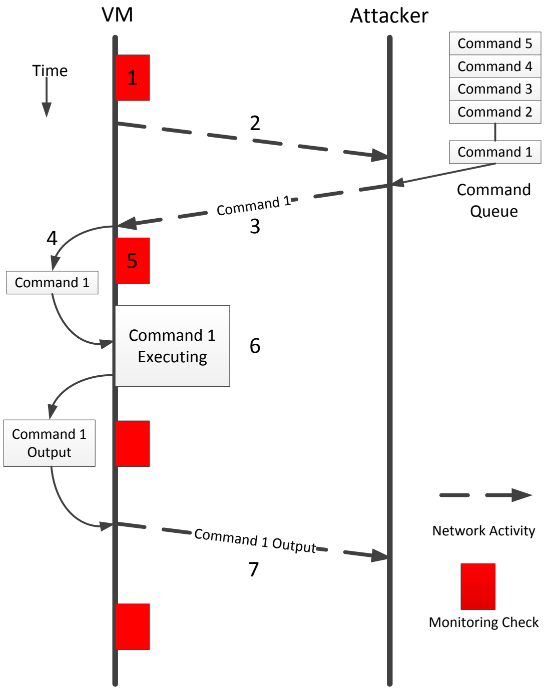


Figure 6: Illustration of the modified command cycle to allow for longer running or more verbose commands

be modified to occur over three monitoring intervals to make the backdoor client more flexible with longer running commands by saving the command output and waiting for the third monitoring check to occur before sending the output back to the backdoor server. The modified command cycle is shown in Figure 6. The modified command cycle would allow for longer running or more verbose commands at the cost of a decreased command throughput (i.e., the backdoor server has to wait longer to receive the command output). However, this tradeoff may be worth the loss in throughput depending on the attacker's goals and requirements.

The command output latency depends on the polling frequency of the VMI monitor similar to the file transfer latency described in Section 4.2. If the polling frequency is low, the attacker will have to wait a longer period before the output of the command is returned to the attacker. The attacker may partially work around this by modifying the command cycle to send more than one command to be run at a time.

5 Defenses Against Hypervisor Introspection

In this section, we discuss current state-of-the-art defenses against side-channel attacks in virtual environments and their shortcomings in defending against HI. We also discuss a potential defense against HI that aims to address the shortcomings of the current state-of-the-art defenses.

5.1 Introducing Noise to VM Clocks

Because HI relies on fine-grained timing measurements to determine occurrences of VM suspends, it follows that reducing the accuracy or granularity of VM clocks could prevent HI. Previous work have looked at reducing the granularity or accuracy of time sources to prevent cross-VM side-channel attacks. Although these previous work aims to address cross-VM side-channel attacks, many are also somewhat applicable to hindering HI.

Vattikonda et al. explored the possibility of fuzzing timers to reduce the granularity of measurements needed for side-channel attacks [13]. They modified the Xen hypervisor to perturb the x86 `RDTSC` instruction by rounding it off by 4096 cycles. Because the `RDTSC` instruction is commonly used to obtain high-resolution timestamps, side-channel attacks may be prevented. An additional benefit of fuzzing the `RDTSC` instruction was that timing system calls, such as `gettimeofday` and `clock_gettime` were fuzzed too. Although HI relies on one of these system calls for timestamping, the perturbations caused only a 2μs change in the true `RDTSC` value. HI needs measurements on the order of milliseconds, so the fuzzing does not perturb the `RDTSC` value enough to hinder HI.

Li et al. developed a system called StopWatch that replaced the VM clock with a virtual clock that would constantly skew [6]. Because the virtual clock only depends on the number of instructions executed, it would hide the VM suspends from in-VM timing measurements. However, applications with real time requirements would not be able to use a virtual clock. Additionally, StopWatch has a worst-case performance of 2.8x for workloads that require heavy network usage, which might not be acceptable for high performance workloads.

5.2 Scheduler-based Defenses

Recently, scheduling policies have been explored as another means to prevent cross-VM, cache-based, side-channel attacks. Varadarajan et al. proposed using the scheduler to enforce each process to run for a minimum run time before another process is allowed to run [12]. By forcing all processes to run for a certain amount of time without being preempted, an attacker would obtain less information from observing a process' cache usage.

Adjusting the scheduling policy could prevent part of our HI technique if the minimum run time is greater than the VM suspend threshold. If that is the case, then process scheduling cannot be used as one of the events observed for the in-VM timing measurements. However, there are many other events that occur during normal OS operation that could still be observed, such as network operations or memory allocation and deallocation. An attacker could also artificially spawn processes that purposely utilize specific OS operations to increase the frequency of the events and improve the granularity of the measurements needed for HI. Thus, changing the scheduling policy may hinder HI, but it would not altogether prevent it. Additionally, enforcing a minimum run time could degrade performance because CPU-intensive workloads would have to compete with less intensive workloads.

5.3 Randomized Monitoring Interval

Because HI looks for regular VM suspends to determine when monitoring checks occur, it may seem that simply randomizing the monitoring interval would prevent HI. This is not the case because randomized intervals have a lower bound on the duration between monitoring checks. A patient attacker could use HI to establish the lower bound on the monitoring interval and craft his attacks around that lower bound. Thus, the randomized monitoring interval forces the attacker to be inefficient, but it cannot prevent the attacker from evading the VMI system.

For example, consider a passive VMI system that polls the guest VM on a randomized interval that lasts anywhere from 1 to 10 seconds, inclusive. Assuming that the monitoring interval is a discrete random variable (i.e., the monitoring interval is an integer), then 10 monitoring checks are expected to be observed before the smallest possible interval is seen.

Even if the monitoring interval is a uniform random variable (i.e., the monitoring interval can be a non-integer value), the attacker can transform the problem into the discrete value case by taking the floor of the observed monitoring interval. In the case of an interval that lasts anywhere from 1 to 10 seconds, flooring the value leaves 10 possible values. Thus, the expected number of observed monitoring checks before obtaining the minimum is still 10.

The range of the random monitoring intervals is directly related to how difficult it is for an attacker to establish the minimum possible monitoring interval. Because security improves with a lower monitoring interval/higher monitor frequency, the range of monitoring interval values would remain small in practice to improve security, but it would also be easier for an attacker to determine the lower bound of the randomized monitoring interval through HI.

We reconfigured our VMI monitor to use a randomized monitoring interval between 0 and 2 seconds that would change after each monitoring check. We chose this interval because it maintained the performance overhead around 5% (based on the same UnixBench benchmarks from earlier) and we expected the lower monitoring intervals (sub-1s) to detect the attacks described in Section 4. We performed each attack 30 times and found that the reconfigured monitor was able to detect the large file transfer 70% of the time, but it was unable to detect the backdoor shell when quick-executing commands were run (e.g., `cat /etc/shadow`).

We reason that the backdoor shell attack is not detected because the command cycle executes quickly enough to remain undetected for most of the intervals from 0 to 2 seconds. We were surprised, however, that some large file transfer attacks succeeded against the randomized monitoring defense. Some of the large file transfers went undetected because the randomized intervals that triggered the chunk transfers in those cases were long enough to avoid detection. However, the majority of the large file transfers were detected with randomized monitoring. Based on these tests, we argue that a randomized monitoring interval is sufficient for preventing the large file transfer attack, but not the backdoor shell attack. Additionally, randomized monitoring does not prevent HI from learning that there is a passive VMI system in place, and HI can be used to learn the distribution sampled by the randomized monitor.

5.4 Non-blocking Monitoring Checks

Another straightforward defense against HI is to make the monitoring checks non-blocking (i.e., do not pause the VM when performing the check). However, non-blocking monitoring checks run the risk of allowing various events to pass undetected due to race conditions. Previous work has looked at using non-blocking checks, only pausing the VM if the non-blocking checks detect an inconsistency [4]. Thus, the non-blocking checks are not guaranteed to extract a consistent state from the VM, resulting in a need to fall back to pausing the VM, which would be detected by HI.

5.5 Proposed Defenses Against Hypervisor Introspection

A virtual clock similar to the one implemented in [6] would work best to prevent the timing measurements needed for HI. A strictly virtual clock could not be used for applications that need a real time clock. Thus, we propose a hybrid real-virtual clock that skews only after a VM suspend. This could be achieved by changing the clock function in the hypervisor after a VM entry occurs. The virtual clock function would skew the clock to catch up to real time before switching back to the real time clock. By using the hybrid clock, there would be no large gaps in the time due to VM suspends and the clock would be near real time.

Instead of attempting to hide the VM suspends of a passive VMI system, an active VMI system may be used [9, 8]. Because an active VMI system only performs a monitoring check after being triggered by some hardware event, there is no way to hide activity from the checks unless the malicious activity does not trigger the hardware event. HI could be used to determine what event triggers the VMI system, but there are many potential events, making determining the exact cause difficult.

6 Conclusion

We presented hypervisor introspection as a technique to determine the presence of and evade a passive VMI system through a timing side-channel. Through HI, we demonstrated that hypervisor activity is not perfectly isolated from the guest VM. Additionally, we showed two realistic attacks in an insider threat attack model that utilize HI to hide malicious activity from a realistic, passive VMI system. After developing HI, we also propose that passive VMI monitoring has some inherent weaknesses that can be avoided by using active VMI monitoring. Thus, future research should continue to focus on the development of active VMI monitoring systems that do not poll the VM, and instead respond to specific events.

References

[1] byte-unixbench - a unix benchmark suite - google project hosting. Accessed: April 20, 2015.

[2] FRANKLIN, J., LUK, M., MCCUNE, J. M., SESHADRI, A., PERRIG, A., AND VAN DOORN, L. Remote detection of virtual machine monitors with fuzzy benchmarking. *SIGOPS Oper. Syst. Rev. 42*, 3 (Apr. 2008), 83–92.

[3] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium* (2003), pp. 191–206.

[4] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with osck. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 279–290.

[5] KORTCHINSKY, K. Cloudburst. Tech. rep., Immunity, Inc., Miami Beach, Florida, June 2009.

[6] LI, P., GAO, D., AND REITER, M. 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.

[7] PAYNE, B. D. Simplifying virtual machine introspection using libvmi. Tech. Rep. SAND2012-7818, Sandia National Laboratories, September 2012.

[8] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (2008), IEEE, pp. 233–247.

[9] PHAM, C., ESTRADA, Z., CAO, P., KALBARCZYK, Z., AND IYER, R. 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.

[10] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. 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* (2009), ACM, pp. 199–212.

[11] SUNEJA, S., ISCI, C., DE LARA, E., AND BALA, V. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2015), VEE '15, ACM, pp. 133–146.

[12] VARADARAJAN, V., RISTENPART, T., AND SWIFT, M. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 687–702.

[13] VATTIKONDA, B. C., DAS, S., AND SHACHAM, H. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (New York, NY, USA, 2011), CCSW '11, ACM, pp. 41–46.

[14] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 305–316.

[15] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 990–1003.