Who Watches The Watcher? Detecting Hypervisor Introspection from Unprivileged Guests

By

Tomasz Tuzel, Mark Bridgman, Joshua Zepf, Tamas K. Lengyel, K.J. Temkin

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2018 USA

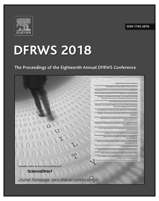
Providence, RI (July 15th - 18th)

Digital Investigation 26 (2018) S98–S106

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin



DFRWS 2018 USA — Proceedings of the Eighteenth Annual DFRWS USA

Who watches the watcher? Detecting hypervisor introspection from unprivileged guests

Tomasz Tuzel*, Mark Bridgman, Joshua Zepf, Tamas K. Lengyel, K.J. Temkin

Assured Information Security, Greenwood Village, CO, USA

Keywords:
Virtualization
Hypervisors
Virtual machine monitors
Cloud computing
Wall timing
Caches
Side-channel attacks
Non-temporal instructions

ABSTRACT

We present research on the limitations of detecting atypical activity by a hypervisor from the perspective of a guest domain. Individual instructions which have virtual machine exiting capability were evaluated, using wall timing and kernel thread racing as metrics. Cache-based memory access timing is performed with the Flush + Reload technique. Analysis of the potential methods for detecting non-temporal memory accesses are also discussed. It is found that a guest domain can use these techniques to reliably determine whether instructions or memory regions are being accessed in manner that deviates from normal hypervisor behavior.

1. Introduction

Cloud computing offers many benefits to organizations of all sizes: economies-of-scale net cost savings, elasticity provides seamless scalability, and consolidation of Information Technology (IT) resources improves service quality and security. To facilitate cloud migration, modern hypervisors aim to minimize the differences between executing in a virtualized environment and on bare-metal by using hardware extensions to multiplex virtual machines (VMs) seamlessly and with minimal performance impact. The hypervisor's position of privilege on the system can come with a negative: a compromised hypervisor is able to introspect and corrupt its VMs, bypassing data protections and giving the adversary control over processing.

The ability of a cloud tenant to detect if and when a host is behaving in an unorthodox or outright intrusive fashion can be valuable in determining whether the platform is to be trusted. As numerous organizations continue migrating services to the cloud, it is essential that software be able to determine the trustworthiness of the environment in which it is executing as well as optimally respond to possible threats.

In this paper, we present our findings regarding the utilization of hardware side-channels to gain insight into computing environments, the limitations of this technique, and the potential for developing a framework to determine optimal responses. Using hardware side-channel information, we have evaluated the feasibility of using shared CPU resources to characterize privileged software. Herein, we provide a body of research regarding the limitations of environmental characterization of virtualized platforms.

Our tool, *Environmental Characterization and Response (ECR)*, analyzes instructions and memory accesses on a guest system which has been deployed on a hypervisor. *ECR* leverages a variety of metrics to determine the potential presence - or lack - of introspection, and serves to establish the limits of attack and limits of detection touched upon earlier. The *ECR* effort developed a novel technology capable of characterizing a cloud platform's privileged architectural software from within an unprivileged environment, providing the foundation for development of autonomous, self-protecting cloud applications.

Our contributions are as follows:

- Provide an in-depth overview of the effects of virtualization on shared hardware resources from a micro-architectural perspective
- Evaluate the efficacy of several timing techniques to supply a robust baseline to build detection systems
- Perform extensive experiments on the capability and limitations of detecting a variety of introspection techniques, including hypervisor accesses to particular in-guest memory ranges, instruction trapping and memory access tracing

* Corresponding author.
E-mail addresses: tuzelt@ainfosec.com (T. Tuzel), bridgmanm@ainfosec.com (M. Bridgman), zepfj@ainfosec.com (J. Zepf), lengyelt@ainfosec.com (T.K. Lengyel), k@ktemkin.com (K.J. Temkin).

https://doi.org/10.1016/j.diin.2018.04.015

2. Related work

The topic of malicious hypervisors has been widely discussed and has produced a significant body of work over the years. From the outset, there have been concerns that adding layers underneath the operating system (OS) will result in systems that may undermine, or outright compromise, the security and privacy of the OS (Rutkowska, 2006; Zovi, 2006).

For the detection of such hypervisors, many techniques have relied heavily on finding implementation-specific artifacts (Ferrie, 2007). Widely available open-source tools today showcase this approach by detecting hardware or software artifacts exposed to the guests by specific hypervisors (Paranoid Fish, 2018). It has also been proposed to utilize even lower layers for detection, such as the System Management Mode (Rutkowska and Wojtczuk, 2008). There has also been research which evaluated the notion of looking for hardware side-effects that a hypervisor would inadvertently introduce to the system (Thompson; Brengel et al., 2016; Fritsch, 2008).

In today's computing environment, however, the existence of a hypervisor is commonplace. Most of the research efforts thus far have not made a distinction between the detection of a hypervisor and the detection of an *introspecting* hypervisor. The research that is available is focused on the evaluation of the stealth attributes of malware analysis systems, such as Ether (Dinaburg et al., 2008) or DRAKVUF (Lengyel et al., 2014). Research into the limitations of these stealth approaches mainly involved looking for specific artifacts, such as discrepancies in the behavior of timing sources as these are being manipulated by the sandbox (Pék et al., 2011).

3. Background

In the following section we provide a brief but in-depth background for the concepts that *ECR* is built upon.

3.1. Virtualization

The creation of a VM that behaves as a typical hardware-based machine running a standard OS, but is separated from the actual physical hardware resources of the hosting system is commonly referred to as virtualization. This technology has lent itself to the birth of the cloud, which offers immense cost-savings through data-center consolidation, centralization of IT, and purchasing power of providers, however, there are security concerns with moving to cloud infrastructure. The most serious of these concerns being the risk of malicious software compromising a hypervisor and utilizing this privileged position to interfere with the operation of VMs, or observe sensitive data in those VMs.

3.2. Hypervisors

A hypervisor is a piece of privileged, low-level software that supervises the execution of guest VMs, and is typically responsible for maintaining isolation between those VMs. To provide a guest experience consistent with running on real hardware, a hypervisor typically shares hardware resources between VMs, directly or indirectly multiplexing access to real hardware resources. There are two generally accepted classifications of hypervisors: type-1 and type-2. A type-1 hypervisor is a bare-metal hypervisor, in which the hypervisor runs directly on the hardware. A type-2 hypervisor is a hosted hypervisor, in which a hypervisor runs as a process on the base OS. In this effort, the Xen Project hypervisor, which is a type-1 hypervisor, was utilized.

Typically in the Xen architecture, the core hypervisor only directly arbitrates access to a few critical system resources, including the CPU and RAM. To mediate access to the remaining hardware, Xen creates a domain known as dom0, which is empowered with the ability to perform hardware access by mapping hardware resources directly into that domain, creating a domain that is uniquely privileged, but which still has significantly less privilege than the virtual machine monitor (VMM) itself. In most use cases, the hardware domain typically runs a standard Linux distribution, such as Red Hat Enterprise Linux (RHEL) or Debian, which provides the drivers used for hardware interfacing and multiplexing software used to route networking traffic to and from the guests.

3.3. Virtual machine exits

To provide an environment capable of executing user workloads that include unmodified system software, hypervisor platforms must be capable of interceding when a guest attempts to perform operations that can impact the state of the real hardware. To enable efficient intercession, processor virtualization technologies, such as Intel's VT-x, provide hardware features that allow hypervisors to assume control once a privileged operation is attempted. As such, the hypervisor software has an opportunity to replace the relevant operation with its own handlers, which often perform equivalent operations to real hardware while limiting scope only to the active VM.

In VT-x terminology, a virtual machine exit, or VM-exit, is a point at which guest execution is paused and execution is returned to the hypervisor, which can then opt to intercede on the guest's behalf. To allow convenient world switches, VT-x based hypervisors use a special-purpose region of memory known as a VM Control Structure (VMCS), which is a data structure consisting of six logical groups that handles hypervisor operations and state transitions between the hypervisor and the guest.

During an exit (Ott, 2018):

1. The cause of the exit is recorded in the VM-exit information fields.
2. The current processor state is saved in the guest-state area.
3. The model-specific registers (MSRs) are stored in the VM-exit MSR-store area.
4. The processor state is loaded from the host-state area and VM-exit controls.
5. The MSRs are loaded from the VM-exit MSR-load area.

Once the hypervisor completes its operations, a VM-entry will be performed to transition control back to the guest. Since the states are stored in main memory, the entire routine results in significant overhead, due to generally low access rates versus a processor cache. This is key to enabling detection of an introspective hypervisor.

3.4. Timers & timing methods

Modern x86-64 platforms contain a variety of timers and timing methods. These include the *x86 Timestamp Counter (TSC)*, the *High Precision Event Timer (HPET)*, and the *Advanced Configuration and Power Interface Power Management Timer (ACPI PMT)*. The first two timers are of interest to us in this paper, and as such, will be addressed here.

The TSC is a high precision timer on-board modern x86 systems which is precise enough to measure individual processor clock cycles. This precision makes it a typical timing source used for analysis of timing-based side-channels, but it is important to note that the TSC value is easily modified by hypervisors. Hypervisors can easily intercede in requests for TSC values, and often do so for legitimate purposes, such as the suspension and resumption of a guest, or malicious reasons such as thwarting attempts at introspection detection. In the *ECR* trust model, where architectural software is not fully trusted, the TSC is a less useful source of timing information and must be used carefully.

Alternative timing sources include the x86 HPET, a timer which is less precise, but is much more difficult to manipulate. As the HPET is commonly used as a source of synchronization for multimedia streams (Intel, 2018b) (wall time), manipulation of this timer by the hypervisor will often cause immediate guest behavioral issues; for example, failure of user interface animations or video playback, and thus is often impractical for an adversary to manipulate. Accordingly, timing sources derived from the HPET can be more easily validated and thus provide more reliable bases for side-channel analysis.

Another strategy for obtaining timing metrics is to execute segments of code with known timing properties (Thompson). These segments are especially hard to identify for modification or emulation, as the halting problem makes the program analysis required untenable. In multi-threaded environments, loops that execute these segments can be used as timing sources by running a loop in parallel with an operation of interest, and counting the amount of iterations that complete. These segments represent an interesting timing source, as they have few hardware dependencies and are resilient to hypervisor manipulation.

Together, the latter two of these timing sources provide a relatively strong basis for analyzing timing side-channels which is simultaneously precise enough for many relevant measurements and which is resilient to modification by privileged software.

3.5. Cache

Access to main memory is slow, and thus, modern CPUs attempt to compensate for the shortfall in latency by storing data and instructions that are likely to be re-used in short order with a small on-processor memory known as a cache. Typically, the organization of a cache is hierarchical, with the L1 cache being small yet fast, the L2 cache being larger and somewhat slower, and the L3 cache being larger and slower than the previous two. The L3 cache is typically inclusive (meaning that data stored in L1 and L2 are also stored in L3), and shared between all processor cores on a multi-core processor, whereas the L1 and L2 are dedicated to a specific core. This configuration can of course vary, however, this is a common arrangement of Intel's caches.

Typical `x86` caches are set associative, thus physical memory is divided into a number of regions called cache sets, within which there exists 64-byte units called cache lines. The number of cache-lines makes up the amount of 'ways' in which the cache is associative. Again, only memory that is likely to be re-used in short order is typically available in the cache, due to the lack of storage in the cache versus main memory. (For `x86-64` processors, the cache is typically 32-kilobytes for both the level 1 instruction cache and the level 1 data cache, 256-kilobytes for the level 2 cache, and 2-megabytes or greater for the level 3 cache.)

Inevitably, all data that resides in the cache will be evicted. In the case of x86-64, the cache replacement policy will result in a given line being evicted only if other blocks from the same memory region are accessed. With this in mind, eviction can be taken as evidence that an access has occurred within a region. Thus, this notion can serve as a basis for eviction-based side-channel attacks.

3.6. Cache side-channel attacks

Keeping in mind that a memory access will populate the processor's cache-lines, it is possible to assess the likelihood that a region of memory is being accessed. Moreover, given that the cache is a shared and unprivileged architectural resource, it thus represents a prime source of side-channel attacks. Given that reading a piece of memory is markedly faster if it is represented in the cache (since the processor does not need to fetch it from memory), timing various operations on a specific portion of memory will make it readily apparent whether that portion is available in the cache. There are complications to this assertion, such as the virtual address to physical address lookup, which must be represented in the translation look-aside buffer (TLB) for minimum timing, although such differences are generally observable.

There have been a significant number of attacks published on this topic (Ge et al., 2016; Gruss et al., 2016; Disselkoen et al., 2017):

- *Prime + Probe*: The attacker primes the cache by populating cache-lines, waits for some time, then probing and timing access to the same cache-line. If timing is lower, it is indicative that the victim has touched that memory, as it was thus already cached.< `\item \emph{Flush+Reload}`: The attacker flushes by shared virtual address, flushing a shared memory-line, waiting for some time, then timing how long it takes to reload the line. A lower timing indicates that the victim is likely to have touched that memory-line. The attack allows targeting of a specific memory-line, as opposed to just the cache set.
- *Flush + Reload*: The attacker flushes by shared virtual address, flushing a shared memory-line, waiting for some time, then timing how long it takes to reload the line. A lower timing indicates that the victim is likely to have touched that memory-line. The attack allows targeting of a specific memory-line, as opposed to just the cache set.
- *Evict + Time*: The attacker waits for some time for the victim to load some cache sets. The attacker can then evict any desired memory-lines, and wait on the victim again. Access to a specific line can be verified based on how long this eviction takes to run.
- *Flush + Flush*: The attacker continuously flushes a shared memory-line, timing the operation. The timing result is used to determine whether the memory-line has been cached. A determination can be made since the victim must load the memory, which would then take longer to flush by the attacker's continuous operations.
- *Prime + Abort*: Leverages features of Intel's Hardware Transactional Memory (TSX) to avoid the use of timers to determine cache accesses. The attacker opens a transaction, accesses the memory address of interest, then waits for an abort. Once the victim accesses the memory, a hardware callback is received through TSX. This attack has a variety of benefits (aside from the need to not utilize timers). It is also markedly faster, does not require predefined intervals (the attacker can passively wait), the timing in which the access occurred is precise (rather than the previous more coarse-grained approaches), and there are fewer false-positives. However, support for TSX may be limited due to its somewhat more recent introduction and checkered roll-out.

It is worth noting that data being represented in the cache does not necessarily indicate that it was accessed recently. The cached location may represent temporal and spatial localities. A temporal locality means that a referenced memory location that was recently accessed is likely to be accessed again in the near future. A spatial locality is the result of a memory location being accessed, thus resulting in nearby memory locations being prepared for possible accesses. With this in mind, it is sometimes better to look for evidence of cache eviction rather than evidence of cache population.

3.7. LibVMI

LibVMI (LibVMI, 2018) is a C and Python library that grants the ability to introspect on a VM's (Windows or Linux) execution on

Xen or KVM. It supports register accesses, event trapping, memory reading/writing, and other features. Full details of LibVMI are encapsulated in its online documentation.

The LibVMI library was of interest in the *ECR* effort as it represents a basis from which many real introspective hypervisors are built, and thus is useful for characterization of malicious hypervisors.

4. Approach

4.1. Motivation

Despite the powerful benefits of virtualization, cloud platforms introduce significant new attack vectors. Such an environment allows an adversary that controls it to use its privileged position to the detriment of the guest instances, should it so choose. Generally, a guest has no choice but to implicitly trust that the provider is behaving as expected, and is providing it with a secure environment.

While the activities of a hypervisor are resistant to direct detection, a hypervisor's operation is not entirely transparent to its guests. A hardware-assisted hypervisor can provide guest software an environment that is functionally equivalent to operation on native hardware, but cannot fully hide its impact on system performance or its utilization of shared, platform-controlled resources, such as CPU caches (Quinn, 2012). These non-functional impacts allow guests a small window with which to glean information about their environment, and can be measured to distinguish between native and virtualized environments for anti-reverse-engineering or rootkit-detection purposes.

Several side-channel measurement techniques exist that measure the non-functional impact of operations that are likely to require hypervisor intervention. If a hypervisor does intervene, it must execute a handling routine from system memory; this execution causes a measurable increase in the number of cycles required to handle the given operation, and often results in hypervisor code and memory occupying space in the affected CPU's caches (Rutkowska, 2004). Any increase in instruction execution time and/or resultant cache artifacts can be observed from within the guest, breaking the illusion of transparency and providing direct evidence of the hypervisor's intervention. Beyond indicating the presence of the hypervisor, this technique can be used to identify events that result in hypervisor intervention, providing valuable hints as to the hypervisor's behavior.

Existing works have successfully utilized side-channel measurements to detect hypervisor impact, but leverage coarse measurements that give an all-or-nothing indication as to whether virtualization is being employed (Quinn, 2012). Measurements at these granularities are of little use in the characterization of cloud environments, where virtualization is a fundamental assumption. To probe the limitations of characterization of cloud environments, existing side-channel ideas needed to be refined to better target hypervisor characterization.

4.2. Implementation

ECR consists of two predominant components:

1. A *test framework*, which emulates various forms of inappropriate introspection and which can emulate malicious hypervisor behaviors to assess our detection ability, and the monitoring module.
2. A *monitoring module*, which resides in the deployed guest instance, and is composed of a set of sensors that attempt to detect a variety of malicious hypervisor behaviors, including inappropriate introspection.

4.3. Monitoring module and sensors

The core challenge in detecting malicious architectural elements, such as a hypervisor performing inappropriate introspection, is that cloud tenants are typically significantly less privileged than the suspect components. To compensate for this, *ECR* has developed a monitoring module that runs entirely from within a guest context, and which requires no additional privilege.

The enabling technology here are side-channel analysis sensors, which use architectural properties to detect hypervisor behaviors. We've developed four primary types of sensors:

1. *Instruction intercession sensors*, which detect when the hypervisor chooses' to intercede in instruction execution and use intercession timings to characterize what the hypervisor is doing in these cases.
2. *Memory intercession sensors*, which detect when the hypervisor chooses to actively intercede in memory access operations. Access timing here is also telling: we can tell what memory the hypervisor chooses to intercede with and how long it takes to intercede.
3. *Passive memory monitoring sensors*, which utilize cache side-channels to identify when hypervisors are inappropriately accessing memory externally to a guest.
4. *Non-temporal access sensors*, which we've prototyped but did not completely implement. The methods developed herein can attempt to detect hypervisor accesses through e.g. non-caching page mappings or using non-temporal instructions.

The core monitoring module itself combines everything from these sensors into a single *health report* that indicates the behaviors observed. This is primarily intended to assess how well the sensors work for scientific purposes, but is a good base for developing a runtime detection framework.

The monitoring module is a single kernel object which creates a device that receives IOCTLs from userspace. This facilitates the ability to have multiple capabilities that the user can specify as needed. This interaction is facilitated by a Bash script, which has been written with the minimal amount of package dependencies that are reasonably necessary in a standard Linux environment to allow for maximum portability.

4.4. Instruction intercession sensors

One of the most primitive, yet most commonly used methods to maliciously affect a guest's operation is to actively intercede in the execution of a variety of instructions using Intel's VT-x technology, which allows the hypervisor to trap on certain instructions. Such a technique is useful in both modifying the guest's execution behavior, determining when a guest performs key operations (such as switching between operations), and extraction of information. It is often the case that a malicious hypervisor must hook certain operations, such as `\emph{MOV to CR3}`, in order to install further hooks (for instance, to hook interrupt handlers, identify when a program is running, or to locate pages of interest). In particular, `\emph{MOV to CR3}` is one of the most commonly hooked operations of an introspective hypervisor, as that is called each time a context switch occurs and provides the location of the running application's root page table, which contains the guest's physical address of all relevant pages.

To identify these intercessions, the duration of instruction executions can be timed. This timing is indicative of two items:

1. *That intercession happened*, since knowing that an instruction took significantly longer, and thus that intercession occurred, is valuable. This is because there are some operations for which

there is effectively no reason a well-behaved hypervisor would need to intercede. These ”red flags” can be used for a quick identification of malicious behavior.

2. *A 'signature' of the handler code*, with which we can get an idea of roughly how much work the hypervisor is doing during an intercession by timing the length of each intercession. These give us both an indicator as to whether the behavior is suspicious (is a hypervisor taking significantly longer than we'd expect for a simple operation) and an indicator of whether a particular piece of code is running.

In the latter case, we can likely predict a range of acceptable timings given the relevant instruction.

To produce a reliable assessment, we need accurate timing sources for which we are confident the hypervisor cannot interpose, keeping in mind that many hypervisors already have simple ways to hide the time they take to interpose (for example, by modifying the read time-stamp counter (*RDTSC*) to only count cycles actively executed within the guest). Accordingly, in addition to common timing sources (*RDTSC*), we have developed a set of timing sources [which were touched upon earlier] that are extremely difficult to tamper with:

- *Wall timing* via the x86 HPET, for which modification would result in anomalous behavior, such as interruptions to multimedia, and would thus be quite apparent to an observer. In order to facilitate this measurement, a kernel module will execute instructions of interest many times in a loop so as to smooth out results and deliver an average (running in the kernel was more desirable than user space because it is then running at a higher execution privilege). This set of loops is run multiple times via a Bash script, to smooth out results. The timing measurement is acquired via calls to the function getnstimeofday (), which is obtaining timing metrics from the wall timer.
- *Thread racing*, in which segments of code with known timing properties are executed in parallel with an instruction of interest, counting the number of iterations that are completed for the instruction. One thread runs the instruction in a loop, while the other thread executes *NOPs* continuously. The *NOP* thread is signaled to start and stop once the instruction starts and finishes, respectively. The amount of *NOPs* that have executed are then logged. This operation is handled in the same way as wall timing (that is, the instructions are looped many times, and the Bash script loops these sets of loops to further smooth out results). The ability of a hypervisor to monitor such activities is impractical due to the very substantial overhead that would be incurred for such an operation.

Using these methods, the sensor module evaluates execution timing for a significant variety of instructions, and reports them back in a table compromised of timing metrics that can be used to identify inappropriate behavior. This information can be used both to identify inappropriate introspection and to reason about the targets of that introspection.

Of these instructions, we would take a particular interest in *RDTSC*, *MOV to CR3*, *RDRAND*, and *RDSEED*. *RDTSC* is an interesting case, as it may be frequently trapped for legitimate reasons, thus obfuscating what the purpose of the hypervisor's activities truly are. Manipulation of *MOV to CR3* would permit the hypervisor to modify the expected location of the page directory and page tables for a given task. *RDRAND* and *RDSEED* are a less interesting case; while trapping them would permit nullifying cryptography, a hypervisor that can perform such a task can also just as likely read the guest's memory directly.

4.5. Active memory intercession sensors

In addition to intercession on instruction execution, hypervisors can choose to intercede when given memory pages are read or written. By marking the hypervisor's virtual to physical mappings (the Extended Page Tables (EPT)) to cause a VM-exit, a guest can be configured to transparently trap to the hypervisor each time a given memory address is accessed. In our experiments this capability was facilitated by LibVMI.

Restricting page accesses gives the hypervisor the capability to perform functionally-transparent modification of read/write/execution values. When a trap is triggered, the hypervisor can choose to emulate, swap pages, or even deny a given access rather than allowing it to occur as expected.

Additionally, restricted page accesses can be used for observation of reads/writes/executes to a given page or page range. This is particularly useful if the hypervisor is trying to extract information about the guest, as the hypervisor may now use access patterns to identify where data is and can be notified when it changes, allowing trivial extraction.

Such intercessions can be identified in much the same way that we'd identify instruction intercession: by timing memory accesses and recording the results. For example, it's a red flag if intercessions occur where we don't expect; additionally, we can get an idea of the scope of the intercession by the amount of cycles that it takes to handle the intercession. Moreover, the same timing restrictions apply, and hence, we can use the same timing sources we developed for detecting instruction intercession.

This type of identification of instruction intercession gives a good idea of what's being ”spied” on with a decent granularity, as the hypervisor specifies intercession at a 4-kilobyte (page-sized) granularity. By walking through a range of pages and checking access times, it is possible to identify which pages are being monitored. The timing is, in addition, very substantial, especially when the world switch is compared to a cache miss. There is little, if any, room to question whether intercession has occurred in these cases.

4.6. Passive memory monitoring sensor

The two previous sections represent the most common and significant forms of intercession; however, a hypervisor that desires to minimize overhead or to be particularly stealthy can also use non-intercessory methods to access guest memory. This is accomplished by mapping a guest's physical pages into contexts other than the guest. While this violates the guest's assumption that its memory remains private, and allows other guests or the hypervisor to directly access the contents of guest memory, it does not indicate when the relevant memory has been accessed. As a result, these passive (non-intercessory) methods do not have a direct timing impact, so the method described in the past two sections won't work, hence, we'll need to use a different side-channel.

As in the wall timing and thread racing timing techniques, the time required to access a memory-line can be used to determine whether there is a potential for introspection. The technique used to achieve this is called `\emph{Flush+Reload}`, which was summarized earlier. A more verbose explanation of the steps involved is as follows:

1. The memory-line of interest is flushed from the cache by the spy
2. The spy waits for an ample period of time so as to allow the victim to potentially access the region of memory
3. The same memory-line is reloaded by the spy, and the access is timed

If the victim did indeed access the memory region, the region would have then been cached again. Thus, in the third step, the time required for the spy's memory reload would be much shorter, since the memory-line is already present. However, if the region was not accessed, it would take markedly longer to reload.

The measurement itself is performed by leveraging LibVMI from dom0 to map a region of memory, then performing a *Flush + Reload* across several memory pages, some of which intersect with the mapping. The deviation in timing for the page(s) of interest will be indicative of introspection; specifically, the time required to access the page will be reduced.

As in the previous section, a reliable and accurate source of timing is required. For our purposes, we used tick timing (*RDTSC*) due to its high accuracy. Since we can test *RDTSC* for instruction intercession, it is then reasonable that once instruction vetting is satisfactory, we can use *RDTSC* with a reduced concern that the hypervisor is manipulating the result. It would be possible to fall back on the HPET for these measurements, although this would bring with it certain trade-offs in accuracy which have not been evaluated in this research.

While this measurement, for the purpose of this work, is conducted entirely using *Flush + Reload*, which has a granularity of the size of a cache line (64-bytes) it may be advisable to use *Prime + Probe* first, as its detection capability is equal to the size of a cache slice (2 MiB), and use *Flush + Reload* subsequently to hone in on the desired memory region. The reason for this is that the *Flush + Reload* operation is computationally more expensive, due to the aforementioned finer granularity.

4.7. Non-temporal access experimentation

Over the course of this effort, the stealthiest possible attacker capabilities were hypothesized, and determined an attack case that avoids all previous detection methods: non-temporal introspection. These attacks use the same principles as passive memory mapping attacks, but avoid cache side-channel behavior. Such attacks take their name from their use of Intel's features for accessing non-temporal data, which is data that is unlikely to be used again, and thus lacks temporal locality.

Two primary methods for detecting non-temporal access were identified:

- Use of Intel's *non-temporal, streaming,* and *vector* instructions, which bypass the cache entirely and thus do not leave a defined side-channel. In experiments, these instructions all have cache-coherence related side effects, which made them relatively straightforward to detect. If a cache-line for a given page is populated, and then a non-temporal instruction is used to read/write it, the processor issues a cache flush to maintain coherence. Thus, a side-channel exists that is similar to what was described in the previous section.
- Use of *Intel's Page Attribute Table (PAT)* allows caching behavior to be specified on a per-page basis. Accordingly, passive mappings can be set to be non-cache-interacting, defeating cache side-channels. Using this technique would create a very stealthy method for introspection. Over the course of this effort, methods for detecting these accesses were evaluated, including looking at average memory access latencies to detect relevant bus contention, but provided only limited detection ability. Further work in this area is necessary.

4.8. Methodology for analysis & classification of results

A variety of methods can be used to determine whether an instruction is exiting. For instance, since *CPUID* must exit in a hypervisor environment, it may be desirable to use it as a baseline against other instructions.

To demonstrate a rudimentary system for determining instruction intercession, the following methodology is used. A variety of machines are tested, and a baseline of instructions which are not exiting is created from the pooled mean and pooled variance. A two-sample *t*-test is then used against each instruction's timing result to determine how much they vary from the baseline. If the t-values are sufficiently high, they can reasonably be flagged as potentially exiting, or at least flagged as being of interest.

4.9. Test framework

To assess the effectiveness of our sensors, we need an interactive test framework that will simulate each behavior of an introspective hypervisor. As *ECR* is targeted towards establishing the limits of detection, the test framework was designed to perform relatively low-impact introspection; these tests were emphasized over representative tests.

The test framework is comprised of the Xen Project as the hypervisor (herein referred to as dom0). Upon installation, the hypervisor then becomes the first component that runs after the bootloader exits. Xen Project was selected due to its wide usage and mature code base, and because it is an open source project. In order to properly develop, analyze, and demonstrate introspection by a hypervisor, Xen Project was modified to accept new hypercalls which would toggle various types of introspection. This capability can be interacted with from dom0. In addition to this, instructions that Xen does not support for VM-exiting by default were also implemented.

With these modifications, the test framework is capable of simulating a wide variety of possible introspection techniques, which thus enables efforts in determining if they could be detected by the guest.

5. Results and discussion

The following sections describe the testing results of each sensor. It is important to emphasize that we are seeking the limits of detection; specifically, we seek to answer questions as to what we can detect, at what granularity, whether there is a likelihood of false positives, and so on.

5.1. Instruction intercession

As described earlier, many of the instructions that trap were given support to do so as part of the *ECR* effort. To simulate the smallest possible impact, the instructions trap and immediately return, thus delivering the minimal possible reasonable timing that would then lend itself towards understanding the limits of detection. Additionally, we can gauge how much actual work the hypervisor is performing upon intercession to some degree, however, more research would be required to have a full understanding of the potential to reliably assess this.

Results of instruction timing findings are presented in two separate batches, since some of the instructions have a much higher timing disparity than others.

Figs. 1 and 2 encapsulates the majority of the instructions that were measured (showing wall timing results and thread racing results, respectively). It can be observed that there is a marked timing difference in all trapped instructions for which the *ECR* test framework facilitated such behavior. Several instructions – *CPUID*, *INVLPG*, *RDMSR*, *WBINVD*, and *WRMSR* – must exit in all situations, and thus only their exiting timing is shown here. *XOR* does not exit, and is shown only as a very fast, non-exiting reference instruction.

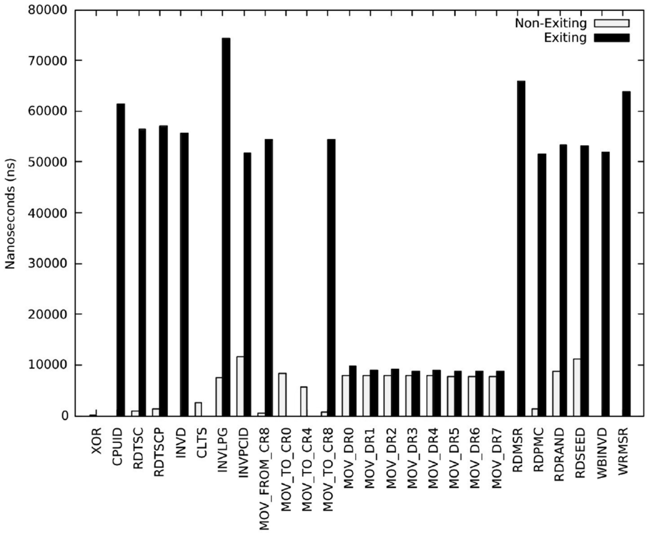


Fig. 1. Wall timing results.

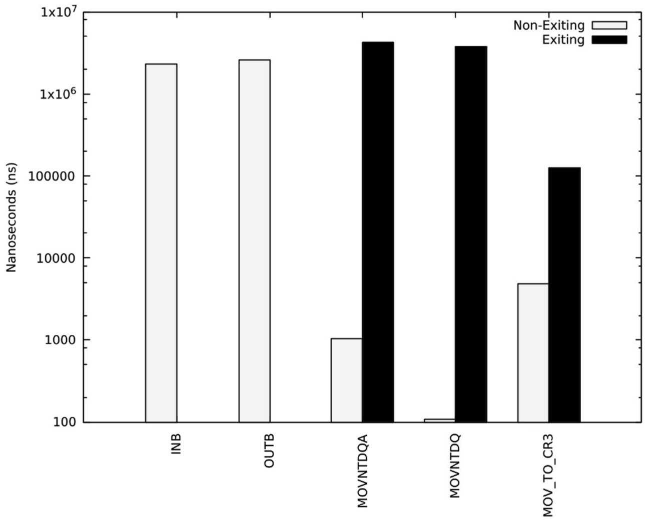


Fig. 3. Wall timing results.

Similarly, *CLTS*, *MOV to CR0*, and *MOV to CR4* also cannot exit (with the hypervisor's existing functionality), and are shown with non-exiting timing as a reference.

Figs. 3 and 4 show the remaining instructions in a log-scale format, which exhibited substantially higher clock timing results. It is important to note that the *MOVNTDQA* and *MOVNTDQ* instructions are not exiting, but rather have their memory accesses handled by the hypervisor's emulator, in the same manner as the memory intercession testing, which is discussed later. *INB* and *OUTB* do not support exiting, and are shown here for reference only.

There is some anomalous behavior with respect to the debug registers (noted as *MOV_DR* in the figures). When exiting is enabled, all of them show a minor increase in timing, but not nearly as substantial as the other instructions. Moreover, only the first register that is tested exhibits a somewhat noticeable increase in timing versus the other registers that are tested. This behavior is due to Xen only trapping on the very first access, then disabling trapping thereafter.

It is clear that in most cases, the measurement results exhibited consistent behavior with regards to very large spreads in timing when being introspected upon and not (when it was possible to toggle such behavior). This lends itself to the conclusion that such an avenue of analysis is viable for a cloud tenant to detect whether a hypervisor is behaving in an atypical manner, since the difference in an exiting instruction versus a non-exiting instruction, even in the case where minimal introspection is performed (again, this was a simple trap and immediate exit), was so large. However, a limitation of this technique is that it does not necessarily lend itself towards determining what an exiting instruction is doing, and to what degree it is doing it. This holds especially true for instructions that must exit by default, since they already exhibit the aforementioned latency increases. Thus, to effectively assess them, it would be necessary to understand what a benign exit would look like. For example, an interesting case of this involves software debugging exceptions involving single-stepping. Whenever a debugger is active in Xen, it traps on this action automatically, so it is not clear whether it is doing this for nefarious purposes or not.

5.2. Active memory intercession

As described earlier, a hypervisor can also utilize EPT to observe guest memory access. This is useful when the hypervisor wants to be

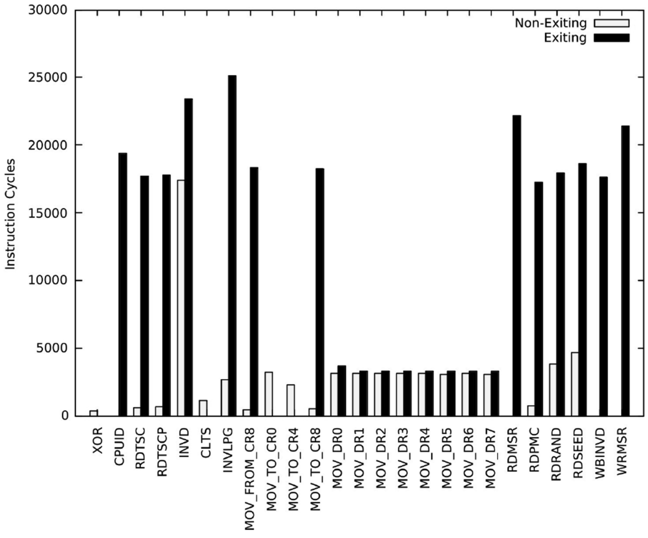


Fig. 2. Thread racing results.

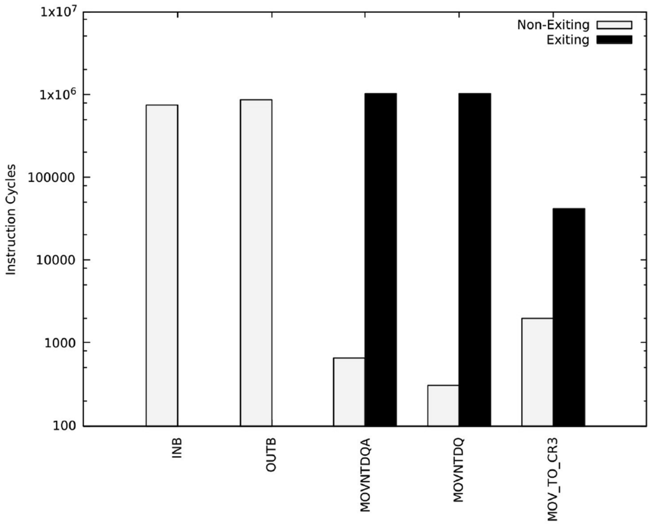


Fig. 4. Thread racing results.

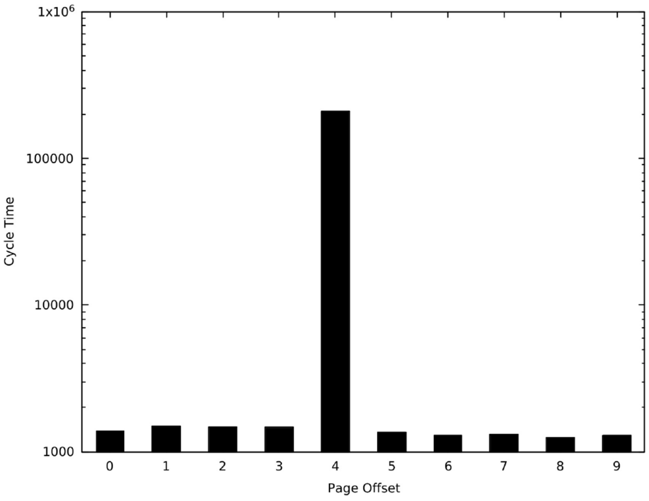


Fig. 5. Memory timing results.

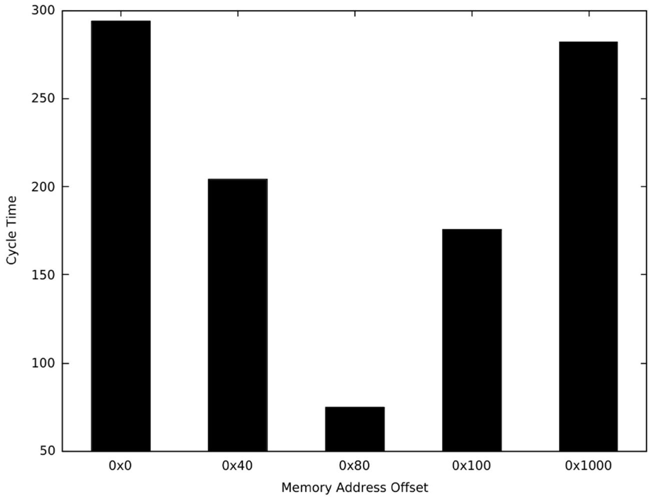


Fig. 6. Cache timing results.

aware of guest access to particular regions of memory with the option of interfering with the outcome of the memory access, which, as an example, can return different data or refuse to write to the address.

In Fig. 5, average results are shown across all tested systems for timed access to multiple memory pages of interest. The page at offset 4 has been mapped by the hypervisor with EPT and marked to cause a VM-exit so the hypervisor may observe the memory before the guest instruction completes. The guest observes a significant increase in access time to page 4. As was described earlier, LibVMI is used to perform an emulated response.

It is, therefore, clear that due to the large disparity in timing of an interceded page versus a non-interceded page, it is unlikely that a guest would observe a false positive.

5.3. Passive memory monitoring

As described earlier, it is expected that the *Flush* + *Reload* operation would complete more quickly when a specific 64-byte cache-line is being mapped. This type of detection technique is most useful in situations where it is desirable to detect passive memory introspection by the hypervisor, which a hypervisor may opt to use when it is desirable to be subtler. While the hypervisor would not be able to easily change information on-the-fly as the guest is using it (as in the active case), it could read and write portions of it as needed, which in many cases, may very well be quite adequate for its purposes.

In Fig. 6, average results are shown across all test systems for an access to multiple memory locations within proximity to one another. In this case, the guest allocated several pages of memory, then performed a *Flush* + *Reload* on several addresses within the allocated range, as depicted by the offsets marked on the X-axis. A program in dom0 utilized LibVMI to access 64-bytes of memory at offset 0 × 80. The guest observes markedly lower timings in this location compared to the other offsets, indicating that this location has been accessed outside the guest (causing it to be unexpectedly cached). This technique provides 64-byte granularity.

6. Future work

The *ECR* effort represents a promising first look into the ability of a cloud tenant to characterize its environment, and provides a solid foundation off of which detection instruments can be developed. There are a number of avenues for follow-on work.

The *ECR* effort establishes the limits of detection and produces simple sensors, but these sensors are not set up for a constantly-running detection environment. The results should be generalized into a continuous-detection framework and integrated into a real cloud tenant.

The *ECR* effort used manual statistical analysis to identify introspection, which would likely be better automated with a full binary classifier. Future efforts could investigate the best classifiers to identify malicious hypervisors.

We did not explore the space of proper responses to identified introspection. Future efforts will need to develop a security model for responding to identified introspection that limits the potential for extraction of data from a guest.

Virtualization Exceptions (#VE) and *VMFUNC*, which are new Intel virtualization extensions, are not addressed here. This is because they are currently regarded as experimental in Xen, and do not appear to be supported in Amazon's EC2 environment, which is the primary environment that *ECR* was oriented towards. These features are geared towards reducing overhead of VM-exits, which would hamper *ECR's* techniques for detecting malicious behavior.

Intel recently introduced EPT-based sub-page permissions (Intel, 2018a), which allows memory protections to extend to a 128-byte granularity, rather than across an entire 4-kilobyte memory page. This would entail a substantial increase in overhead to determine whether a hypervisor is monitoring memory regions.

7. Conclusion

This work demonstrates a reasonable methodology for determining the existence of an introspective hypervisor from within the guest. The ability to reliably determine instruction intercession, as well as active and passive memory intercession was discussed. Detecting instruction intercession for cases in which Intel VT-x supports it, even while using the minimal possible intercession techniques, is clearly evident, given the wide disparity in timing between an interceded case versus a non-interceded case. There is further work required to determine specifically how feasible it is to determine the degree of intercession that the hypervisor may be undertaking, and to propose a response to such intercession.

In the case of memory interaction, both active intercession and passive monitoring detection cases have been demonstrated. These techniques have been shown to work reliably, regardless of most hypervisor attempts as subtlety, and could be expanded to actively monitor for atypical behavior in a real-world environment. This would likely involve determining memory regions of interest, and creating a capability to monitor them by actively rotating through each one, although the proper implementation technique would be subject to some investigation. Moreover, to make a more concrete determination as to whether introspection was occurring, it would be necessary to minimize the possibility of another process accessing the region, which could be done by ensuring that the region is allocated across more than just one page, and to create a situation in which one can be reasonably certain that the page is not being accessed by another process.

Further complicating analysis could stem from a malicious hypervisor attempting to use both active and passive memory introspection techniques. This would lead to an interesting situation, because the result of the activities of one form of introspection would increase latency, while the other would decrease latency, as discussed earlier. The result could lead to obfuscated timings.

This research has wide applications when determining malicious behavior in a cloud environment. Moreover, it serves to establish limitations of this type of detection, showing that certain types of introspection cannot reasonably be conducted without much more serious efforts, or without major performance implications.

Acknowledgements

The authors would like to thank Jacob Torrey for his thoughts and aid in this project. This research is based upon work supported in part by the Office of the Director of National Intelligence, Intelligence Advanced Research Projects Activity (Grant no. 2017-16120700002). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

Brengel, M., Backes, M., Rossow, C., 2016. Detecting Hardware-assisted Virtualization. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 207–227.

Dinaburg, A., Royal, P., Sharif, M., Lee, W., 2008. Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. ACM, pp. 51–62.

Disselkoen, C., Kohlbrenner, D., Porter, L., Tullsen, D., 2017. Prime+abort: a timer-free high-precision l3 cache attack using intel tsx. In: 26th USENIX Security Symposium (USENIX Security 17), (Vancouver, BC), pp. 51–67.

Ferrie, P., 2007. Attacks on More Virtual Machine Emulators. Symantec Technology Exchange, p. 55.

Fritsch, H., 2008. Analysis and Detection of Virtualization-based Rootkits. Munchen: Technische Universitat.

Ge, Q., Yarom, Y., Cock, D., Heiser, G., 2016. A survey of microarchitectural timing attacks and counter measures on contemporary hardware. J. Cryptographic Eng. 1–27.

Gruss, D., Maurice, C., Wagner, K., Mangard, S., 2016. Flush+flush: a fast and stealthy cache attack. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 279–299.

Intel, 2018a. Architecture Instruction Set Extensions and Future Features Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

Intel, 2018b. Ia-pc Hpet (High Precision Event Timers) Specification. https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf.

Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A., 2014. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference. ACM, pp. 386–395.

LibVMI, 2018. https://github.com/libvmi/libvmi.

Ott, D., 2018. Virtualization and Performance: Understanding Vm Exits. https://software.intel.com/en-us/blogs/2009/06/25/virtualization-and-performance-understanding-vm-exits.

Paranoid Fish, 2018. https://github.com/a0rtega/pafish.

Pék, G., Bencsáth, B., Buttyán, L., 2011. Nether: In-guest detection of out-of-the-guest malware analyzers. In: Proceedings of the Fourth European Workshop on System Security. ACM, p. 3.

Quinn, R., 2012. Detection of Malware via Side Channel Information. State University of New York at Binghamton.

Rutkowska, J., 2004. Red Pill: Detect Vmm Using (Almost) One Cpu Instruction. http://invisiblethings.org/papers/redpillhtml.

Rutkowska, J., 2006. Introducing Blue Pill', the Official Blog of the Invisiblethings Org, vol. 22, p. 23.

Rutkowska, J., Wojtczuk, R., 2008. Preventing and Detecting Xen Hypervisor Subversions. Blackhat Briefings, USA.

Thompson, C., Huntley, M. and Link, C. [n.d.], 'Virtualization detection: New strategies and their effectiveness'. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.7877&rep=rep1&type=pdf

Zovi, D.A.D., 2006. Hardware Virtualization Rootkits. Black Hat 2006. August.