Pitfalls of virtual machine introspection on modern hardware

Tamas K. Lengyel
Technische Universität München
tklengyel@sec.in.tum.de

Thomas Kittel
Technische Universität München
kittel@sec.in.tum.de

George D. Webster
Technische Universität München
webstergd@sec.in.tum.de

Jacob Torrey
Assured Information Security
torreyj@ainfosec.com

Claudia Eckert
Technische Universität München
eckert@sec.in.tum.de

ABSTRACT

Over the last few years there has been immense progress in developing powerful security tools based on Virtual Machine Introspection (VMI). VMI offers unique capabilities which can be used to check and enforce security policies in the presence of a potentially compromised guest. With the introduction of new hardware virtualization extensions, VMI can be further enhanced to provide lightweight, in-band control over the execution of virtual machines. In publications released before the extensions were available, security researchers issued warnings that these new extensions may be used to subvert VMI. Since hardware supporting these extensions is now available, in this paper, we aim to discuss and re-evaluate claims made in prior-art. We further continue the discussion by highlighting critical limitations of the virtualization extensions. We go on to show that thorough consideration and understanding of these limitations is necessary when developing VMI based security applications. Otherwise, improper handling will inadvertently expose these applications to subversion attacks. Finally, we take a look at Intel's normal and dual-monitor System Management Mode and discuss how they can be used to both implement and subvert VMI based security applications.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Virtual Machine Introspection, Virtualization, System Management Mode

1. INTRODUCTION

Over the last decade, significant research and development efforts have been made to create out-of-band security systems that leverage virtualization techniques. One of the unique features virtualization offers is the ability to observe a running operating system from an outside perspective, otherwise known as Virtual Machine Introspection (VMI). By further merging the capability with forensic memory analysis (FMA) techniques [7], VMI is rapidly becoming a cornerstone of cloud-security.

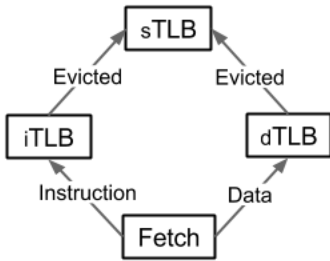
A common problem that security applications face is how to appropriately interact with data-sources that may have been tampered with. The problem is well known when using FMA techniques as they often rely on information contained within the malicious guest OS's memory. As has been shown over the years, the OS under VMI monitoring can be subverted to break the assumptions made about the OS's internal behavior [3, 10]. These attacks are widely known and have recently been named the *strong semantic gap problem* [12].

A common characteristic of the above mentioned attacks is that they are all software based attacks, aiming to subvert the reconstruction of high-level state information regardless of what hardware was used. While prior art briefly discussed hardware based attacks [3], the literature on the topic is sporadic. Altough the information is generally available in the hardware manuals, critical limitations are only mentioned passingly or are only implied when carefully read. Therefore, in this paper we aim to survey subversion attacks that are rooted directly in the hardware virtualization extensions. While some of the subversion attacks we will discuss are academic, their relevance is based on our experience while developing hypervisors and VMI tools.

On modern hardware there are a plethora of new virtualization features that offer unique capabilities for enhancing speed, and also to establishing in-band VMI security systems [5, 26, 14]. Most notable are Intel's Extended Page Table (EPT) and Virtual Processor Identification (VPID) extensions. In the following we will examine these new extensions, their potential utility for VMI, and show in-detail how their shortcomings can have unintended side-effects on security applications aiming to make use of modern hardware. Afterwards we turn our attention to Intel's System Management Mode (SMM) and discuss how this mode can be used to both implement and subvert Virtual Machine Monitor (VMM) based security systems.

2. A CLOSER LOOK AT THE TLB

In the following section we examine the system cache known as the translation lookaside buffer (TLB). We begin by briefly outlining the architecture and how it has been used in the past for offensive purposes. We continue by discussing how changes in recent Intel CPUs affect these techniques and evaluate the impact on VMI applications. Afterwards, we take a look at the new tagged TLB feature and discuss how



2) TLB Entry

Entry: VM	VPID: x	GVA	GPA
Entry: VMM	VPID: 0	VA	PA

Figure 1: Overview of the split and tagged TLB architecture.

modern open-source hypvervisors implemented support for it.

2.1 The split TLB

As virtual-to-physical (V2P) address translation is expensive, even with hardware acceleration, modern CPUs maintain a TLB, which is a transparent cache to store the translation results. To further improve performance, Intel implemented a split TLB architecture which separates the cache into two disjoint sets. The iTLB stores translations for instruction fetches and the dTLB stores translations for data fetches. In newer CPUs, Intel added a secondary cache called the sTLB, which stores the evicted entries from the iTLB or dTLB to offer even faster address resolution. An overview of the TLB system is shown on Figure 1.

The split TLB architecture has been used both for defense and offense. The first system to make use of the split TLB was GRSecurity's PAGEEXEC feature in which they tackled the problem of marking a page non-executable without explicit hardware support for it (like the NX-bit in later CPUs) [9]. In recent years it has also been proposed and used in similar fashion to ensure the integrity of code which resides on pages that mix code and data segments [16, 20].

On the offensive side, the Shadow Walker rootkit leveraged this architecture for stealth purposes [18]. The rootkit took advantage of the fact that a virtual address can point to different physical addresses based on which TLB is utilized. In such a split, the rootkit's code can safely execute without antivirus software being able to scan its code pages [28]. To further make the rootkit more persistent against TLB flushes, the rootkit's code pages are also marked as global. The algorithm to perform this TLB poisoning is shown in Table 1. The poisoned global pages can only be flushed if the TLB is full and the entry is evicted by the hardware, or by turning off and on the Page Global Enabled (PGE) bit on the CR4 register. For example, Windows 7 actively and frequently flushes the global pages from the TLB and disabling the routine leads to a system crash almost imme-

Input: Splitting Page Address (addr), Pagetable Entry for addr (pte)
1. `invalidate_instr_tlb` (*pte*);
2. `pte = the_shadow_code_page` (*addr*);
3. `mark_global` (*pte*);
4. `reload_instr_tlb` (*pte*);
5. `pte = the_orig_code_page` (*addr*);

Table 1: TLB poising algorithm as described by [3].

diately. This behavior effectively limits the life-time of the poisoned TLB. Linux on the other hand does not perform any such TLB flushes, making it a more potent target for TLB poisoning.

In recent CPUs (Nehalem and newer), a secondary victim-cache has been added to the Intel architecture: the sTLB. The sTLB holds all entries which are evicted from either the iTLB or the dTLB, and in the event of a TLB-miss, the sTLB is checked before the system performs a real address translation. If the sTLB contains a matching TLB entry, it is brought back into *both* the iTLB and the dTLB. This further complicates the development of stealthy rootkits, as the sTLB can only hold one version of the evicted TLB entries. For example, if the dTLB entry is evicted and then a data fetch is performed with its virtual address, the entry from the sTLB will overwrite the iTLB entry as well. This poses a particular problem to a rootkit leveraging this technique, as its custom #PF handler code is not invoked to re-split the TLB. This leads to either a non-hidden rootkit code-page if the iTLB entry is brought back from the sTLB, or a system crash / infinite loop if the dTLB entry is brought back and is being accessed as if it was code.

As we can see, malicious code running within the guest on modern hardware is unable to leverage the split-TLB for hiding malicious code from other applications running within the guest. However, with a little help from the VMM, the behavior of the CPU can be adjusted to skip entries being evicted into the sTLB. Unlike regular page-table entries, EPT entries allow setting a page to execute-only; that is, it can be accessed only by code-fetching. When the CPU detects that the iTLB and dTLB have different EPT permissions (one with R/W for data and the other with X only for code), evicted entries skip the sTLB. Thus, if the in-guest TLB-split routine is created by or in coordination with the VMM, the sTLB can be by-passed so that the address translation goes through the primed page-tables again [21] and effectively enabling again the Shadow Walker technique.

2.2 The tagged TLB

In the first implementations of Intel VT-x, the TLB entries were completely flushed during VMENTRY and VMEXIT operations. As a side-effect, this provided a good security counter-measure to a Shadow Walker style TLB technique for hiding in the guest. With newer VT-x implementation, the concept of the tagged TLB has been added to Intel, dubbed VPID. With VPID, the hardware does not flush TLB entries during VMENTRY and VMEXIT. This is because the CPU can now distinguish between entries based on the assigned tag. This naturally results in significant performance boosts on modern hardware.

In a footnote Bahram et. al. [3] speculated that with tagged TLB being available, hiding in the TLB will reemerge

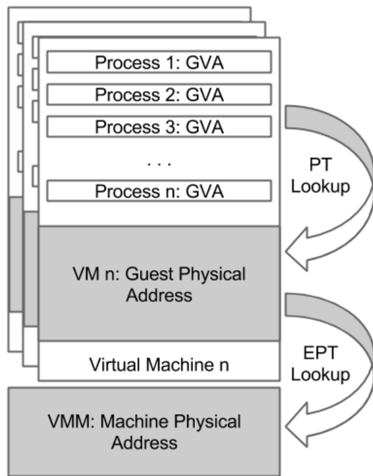


Figure 2: EPT Overview

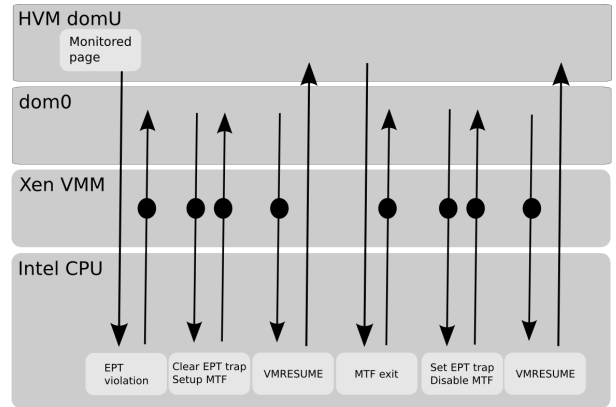


Figure 3: Handling an EPT violation on Xen.

as a method of achieving stealth against VMI applications. However, at the time no hardware was available with tagged TLB support. In contrast, nowadays most modern Intel CPUs have both the sTLB and the VPID feature. As we already discussed, the introduction of the sTLB already affects how TLB-splitting can be performed, which consequently diminishes the utility of the split-TLB as a technique to hide malicious code in a guest. However, hiding from VMI doesn't necessarily require malicious code to use a split TLB. For VMI applications, the TLB itself is the problem, as VMI always emulates address translation in software using the page tables. Any translation cached in the TLB that is no longer reflected in the page tables is effectively invisible to external monitors. In the following, we aim to highlight the different VPID implementations available in modern open-source hypervisors to highlight how these implementations affect VMI applications.

The Intel VPID is a 16-bit field included in the Virtual Machine Control Segment (VMCS) for each vCPU. The assignment of tags is left to the hypervisor with the exception being that tag 0 is a magic tag specifying the VMM. The tagged TLB entries can be flushed by specifying the tag, flushing all tagged entries, or assigning a new tag to the vCPU so the hardware will eventually evict the stale tags. While the description of the tagged TLB is straight forward, the actual implementation in open-source hypervisors varies greatly.

Xen implemented the VPID as a round-robin counter, where a tag is assigned in the VMCS simply by incrementing the counter. When an overflow occurs, all TLBs are flushed, and the iteration restarts at 1. During regular operations, the tagged TLBs are never flushed, instead a new VPID is assigned to the vCPU. As the comment describes it in the Xen source: "Each time the guest's virtual address space changes (e.g. due to an INVLPG, MOV-TO-CR3, CR4 operation), instead of flushing the TLB, a new [VPID] is assigned. This reduces the number of TLB flushes to at most 1/#[VPID]s. The biggest advantage is that hot parts of the hypervisor's code and data retain in the TLB". However, on a closer look at the source we see that the comment is only partially true: the VPID does not get invalidated on MOV-TO-CR4. Additionally, with new VPIDs being assigned on MOV-TO-CR3, the hypervisor effectively disables the guest's ability to use global pages.

If we recall, the purpose of having global pages is to make the TLB entries survive a MOV-TO-CR3 so that the translation can be shared across processes. On Xen, the hardware won't be able to utilize global TLB entries as the VPID tag of the global page is stale after the context switch. As a side-effect, the TLB priming would have to be performed on every context switch.

On KVM the VPID implementation is done by using a bitfield. When KVM creates a new vCPU structure, the first unused bit is reserved to this vCPU. As a peculiar decision, the tag is assigned even to vCPUs that never execute; that is, it is assigned during the creation of the vCPU, not during the first VMENTRY. When the VMM runs out of available tags to assign, it simply disables VPID in the VMCS. This implementation means KVM guests can prime the TLB with global pages without having to perform this operation on every context switch.

3. A CLOSER LOOK AT EPT

In the following section we take a closer look at the EPTs from a VMI tracing perspective. We briefly introduce the extension and how it is used for great effect in various VMI applications. Afterwards we turn our attention to the limitations of the extension and discuss various pitfalls that need to be taken into consideration when building security applications relying on the extension. While these limitations do not automatically break VMI applications, without proper handling they could lead to subversion attacks. While developing real-world security applications the authors have encountered these limitations. In this section we aim to clearly spell out the implications for future VMI developers and security researchers.

3.1 Overview of EPT

Historically, a VM was required to perform a software based address translation from Guest Virtual Address (GVA) to Guest Physical Address (GPA). The hypervisor performed this action through the shadow page table and it had a significant performance cost. In response, Intel introduced EPT, a new virtualization extension. This extension rec-

tified the situation by providing hardware assisted address translations at both stages, as shown in Figure 2. With EPT, the VM no longer needed to invoke the hypervisor to perform page table operations. This provided a boost in performance by alleviating the necessity to perform a VMEXIT when doing an address translation and by freeing the hypervisor from having to maintain the shadow page table.

Security software running outside of the VM has a long history of using the second stage translation to trigger traps for *active* monitoring. This technique was first used by Ether [6] to trace system calls through modifying the access permissions in the shadow page tables. In newer systems, such as CXPinspector [26], the EPT itself has been used for this purpose because violations in the second stage translation traps into the hypervisor. Combined with other CPU extensions, such as the eXecute-Never (NX) bit, EPT allows for tracing arbitrary memory R/W/X operations.

During an EPT violation, the VMCS further describes the location of the violation, both as a GPA and as a GVA. Additionally, the VMCS also describes if the violation occurred during a first-stage GVA translation (violation during the CPU's first-stage page-table lookup) or with the final GPA obtained from the translation.

While tracing memory accesses with EPT is stealthy, the performance overhead is considerable. Each violation on a monitored page needs to be first cleared and then reset to allow the VM to continue the execution but to still catch all subsequent events. On Figure 3 we show the common handling of EPT violations on Xen. In case only a certain section of a page is of interest, the tracer also needs to filter unrelated violations, further adding to the performance overhead and complexity of handling EPT violations.

3.2 Catching modifications

Now that we have a brief overview of how EPT violations can be used to trace memory accesses performed by a VM, we aim to highlight the limitations of EPT through some examples. While the limitations are not necessarily prohibitive, without proper consideration while developing security applications, they can result in a knowledgeable in-guest attacker hiding from 'naive' protection schemes.

Direct Kernel Object Manipulation (DKOM) attacks are a well-known way of breaking both FMA and VMI tools. DKOM works by modifying data-structures in the kernel's heap. The classic example of this is by hiding a malicious process by unhooking its data-structure from the linked-list that the OS uses to report active processes to tools such as *ps*. As the linked-list is a non-critical and independent structure from what the OS uses to schedule processes, this type of DKOM attack breaks the assumption that the list is well maintained and accurately describes the list of active processes.

In the context of detection of unhooked processes, a security application can trace when updates are made to the linked list via the EPT. This is done by checking the violation information contained in the VMCS to see whether it is at the offset of the pointers *next* and *prev*, as a security application has direct knowledge of the updates made to the linked-list. Such an approach may seem straightforward because during normal operations the offset at which the violation is reported matches the location of the pointers. That is, the operating system updates the pointers *directly*. However, a critical limitation in the way the hardware reports EPT violations needs to be taken into consideration in this scenario. During an EPT violation, the CPU only records the starting address where the violation occurred on the monitored page. However, the violation may involve a read/write operation up to 8-bytes. This short-coming is known by Intel, as they already patented the solution [22]. As such it may be the case that some future CPUs will provide this information as well.

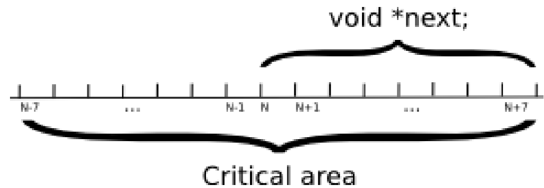


Figure 4: The critical memory region at which EPT violations may occur that could mean an access to the protected region (void *next).

If an attacker knows that only the exact addresses are going to set off an alarm and the rest are filtered as unrelated violations, it then becomes possible to perform a successful DKOM attack. The only task is to break the assumption that the violation will be at exactly the pointer locations. Figure 4 highlights the entire critical memory region that security applications looking for EPT violations need to consider. For example, overwriting the pointers in two steps could perform the attack: first, write 8-bytes starting at (N-1); second, write 1-byte starting at (N+7). The DKOM attack will still trigger VMEXITs, but our naive protection scheme would disregard the violations as unrelated write-events.

3.3 Catching data-leaks

Now we will turn our attention to another potential security feature that EPT could be used for: data-leak prevention. In data-leak prevention systems, it is crucial that specific memory locations are accessed only under certain circumstances. An external security application can potentially use EPT's read protection to enforce a mandatory access control system. The limitation described in the previous section is applicable under this scenario as well. However, EPT has another crucial limitation which could be utilized to siphon protected data without triggering an alert.

The limitation is in how EPT violations are reported when a read-modify-write (r-m-w) instruction is executed. According to the Intel manual: "An EPT violation that occurs during as a result of execution of a read-modify-write operation sets bit 1 (data write). Whether it also sets bit 0 (data read) is implementation-specific and, for a given implementation, may differ for different kinds of read-modify-write operations" [11]. The implications of this behavior are apparent: any memory event subscriber solely looking for read EPT violations as the trigger for enforcing an access control system can be subverted by employing r-m-w operations instead to access the data.

Despite the fact that such unpredictable behavior surrounds these operations, current hypervisors make no attempt in mitigating it in software. For example, up until recently Xen simply forwarded the violation information from the hardware to memory event subscriber applications. Only with our recent patch does Xen mask the hardware limita-

tion from applications by unconditionally marking all write violations also as read violations.

3.4 Virtual DMA and emulation

Thus far we have looked at the hardware limitations of using EPT and its effects on VMI application. In the following we discuss two more scenarios where EPT protected memory regions can be accessed without triggering EPT violations, based on a recent discussion thread on the xen-devel mailinglist [13].

In modern virtualization environments device emulation is a critical component in allowing off-the-shelf operating systems to run without requiring it to be virtualization-aware. For example, Xen uses QEMU to provide various emulated device backends for Windows (or Linux) virtual machines, such as VGA, disk, and network devices. Device emulation however is known to be complex and error-prone, thus being a fertile ground for various exploits. For example, it has been used in the past for breaking out KVM virtual machines [8].

As to mitigate the risk involved in running the QEMU instance in the Trusted Computing Base (TCB), Xen introduced the concept of stub-domains, where the QEMU instance is running in a light-weight paravirtual VM next to the main VM it provides emulation for. Thus, even if an attacker breaks out of the VM via the emulated devices, it only gains access to another de-privileged VM. Nevertheless, such break-outs are not without consequence from a VMI perspective.

On Xen, even the de-privileged QEMU stub-domain requires Direct Memory Access (DMA) into the main VM in order to provide the I/O emulation it is tasked with. In a hypothetical break-out where an attacker successfully compromised the QEMU stub, the DMA functionality can be used to by-pass any type of EPT traps set on the main domain. That is because the stub can request any memory page of the main VM to be mapped into its own address space by the hypervisor. However, the stub being a paravirtual domain, does not use EPT to access the same memory.

Similar problems can be potentially found with the emulated interrupts and timers. For performance and scalability reasons with many-vcpu guests, Xen provides a fast-path emulation for a variety of interrupts and timers, such as RTC, PIT, HPET, PMTimer, PIC, IOAPIC, and LAPIC. Since the emulation happens within the hypervisor, any updates to pages with EPT traps, that happen as a result of emulation, avoid triggering these traps.

4. A CLOSER LOOK AT SMM

In recent years there has been a number of attempts to move VMI to the SMM. However, SMM poses particular challenges for VMI applications, especially when faced with non-cooperating or malicious VMMs. In the following we present a quick overview of the SMM and discuss potential problems that have been described in recent literature in utilizing it as a platform for VMI. Finally, we present an overview of Intel's new dual-monitor mode SMM and how it relates to the outlined problems.

4.1 Normal-mode SMM

SMM, according to the Intel manual [11], was designed to provide an alternative, transparent operating environment to chipset manufacturers and BIOS vendors. This mode can be used to monitor and manage various critical system resources for more efficient energy usage, control system hardware, and respond to thermal emergencies in the event of a non-responsive OS. It is highly privileged and cannot be interrupted by regular interrupts (including non-maskable interrupts), thus guaranteeing the execution of the SMM code once it is triggered. Furthermore, code running within the SMM has full access to the system and can perform arbitrary modifications to the system RAM. In recent years the SMM has received considerable attention from security researchers; for example, it has been shown that SMM could also be used to implement stealthy rootkits [19, 25], VMM integrity verification systems [17, 23] and full-scale VMI applications [29].

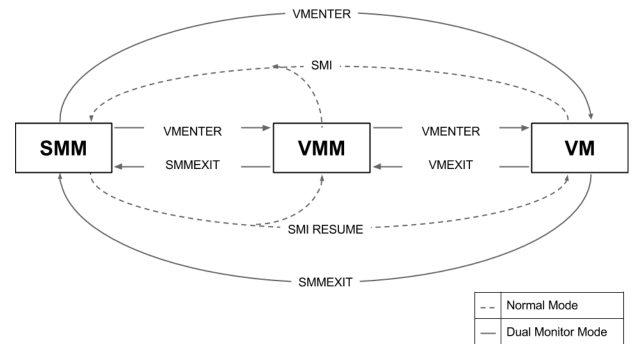


Figure 5: Overview of relationship between SMM, VMM, and VM.

In order to trigger the execution of code within the SMM, a System Management Interrupt (SMI) is issued via the local Advanced Programmable Interrupt Controller (APIC) or via the SMI# pin. In practice the APIC method is the preferred trigger mechanism as the SMI# pin requires extra hardware to be attached [4]. When the interrupt is received, the active CPU context is saved into a dedicated memory region, known as System-Management RAM (SMRAM). The chipset can be configured to trigger an SMI on a multitude of possible events, from USB activity, writes to certain I/O ports, thermal events, and even periodically [24]. When the SMI handler (which executes from within SMM) finishes executing, the previous interrupted operating mode is restored from the snapshot that was saved into SMRAM. An overview of this operation is shown in Figure 5 with the dotted lines. While the SMI handler can modify the CPU register values that were saved, the CPU always returns to the same operating mode that was active when the SMI was received: VMX root or VMX non-root.

As pointed out by Jain et al. [12]: "A limitation of any SMM-based solution [...] is that a malicious hypervisor could block SMI interrupts on every CPU in the APIC, effectively starving the introspection tool. For VMI, trusting the hypervisor is not a problem, but the hardware isolation from the hypervisor is incomplete.âĂİ While it is true that the VMM could starve the SMM in such a way, nothing prevents the SMM from modifying the VMM's code to remove such code-paths. As the SMM is initialized as part of the BIOS before the VMM, any protection the VMM may attempt to leverage against the SMM could be potentially circumvented.

4.2 Dual-monitor mode SMM

The Intel manual also describes an additional CPU mode referred to as *dual-monitor mode SMM (DMM))*. DMM was created to prevent or limit the damage from the exploitation of vulnerabilities in SMI handlers to gain control of SMM. An example of such an attack can be found in [27]. While the original idea behind this implementation was to compartmentalize the SMI handlers into less privileged SMM VMs [15], the capabilities of the hypervisor running in SMM, the SMM Transfer Monitor (STM), in our opinion far exceed what would be warranted.

The primary difference between the two SMM modes is that in dual-monitor mode the SMM enters VMX root mode itself. In this mode, the STM is allowed to create virtual machines within the SMM to run the SMI handlers. In comparison, AMD and ARM simply enabled the regular hypervisor mode to trap SMIs, thus being able to compartmentalize SMI handlers into regular VMs [1, 2].

With the introduction of this new mode, the behavior of the VMCALL instruction has also been extended. On CPUs without DMM support, the VMCALL instruction is only valid if executed in VMX non-root - that is, in a virtual machine. On CPUs with dual-monitor mode, the VMCALL instruction is valid even in VMX root and triggers an unconditional transfer into the SMM.

This behavior has particular importance when we consider how a VMM would attempt to starve the SMM, as we discussed for normal-mode SMM. To recall, in normal mode the APIC could be configured by the VMM to starve the SMM. Now with a dedicated instruction it is no longer possible, as the instruction unconditionally triggers the switch without relying on the APIC to trigger the required interrupt.

As this instruction is no longer pre-emptible from the VMM, it can also be utilized by the STM to position itself inline into the execution of the rest of the system. The STM could inject this instruction as a trap to trigger the transfer of control when code-paths of interest are executed within the VMM. While the presence of these hooks may be detectable if the VMM performs code-integrity checks on itself, the SMM could disable such integrity checks. While this instruction cannot be used to instrument virtual machines - as it would trap into the VMM - it can be used to hook the VMM's trap handlers. Afterwards, the STM can use any of the regular instrumentation methods the VMM has over VMs, thus attaining total control over the execution of the entire system.

Another particularly powerful addition in the dual-monitor mode is that the SMM can jump into any of the other execution modes of the system. In a non-DMM configuration, execution is returned to the same mode that was active before the SMI was triggered whereas in DMM, there are no such restrictions. For example, this mechanism allows the SMM to schedule the execution of VMs that the VMM deliberately suspended. Interestingly, the *stm* can also specify if further SMIs should be blocked for the duration of a VMENTER: "VM entries that return from SMM and that do not deactivate the dual-monitor treatment may leave SMIs blocked. This feature exists to allow an SMM monitor to invoke functionality outside of the SMM without unblocking SMIs" [11]. This design decision is particularly interesting, as now even SMIs can be blocked.

While Intel is unclear about which CPUs support this feature, in our experience it is supported in all CPUs with the VT-x extension. In the VirtualBox source-code we also find a comment for the definition of an MSR [1] that suggests that in their experience this is also the case: "Whether the processor supports the dual-monitor treatment of system-management interrupts and system-management code. (always 1)". Unfortunately, Intel generally locks access to the SMM. To gain control of the mode without Intel's approval would require an exploitable vulnerability, as has been demonstrated in the past [27]. Nevertheless, the dual-monitor SMM mode has many advantageous features that would aid the development of SMM-based hypervisor-agnostic VMI applications. On the other hand, it would also be a prime location to implement applications to subvert VMM-based VMI applications.

5. CONCLUSION

In this paper, we have revisited VMI subversion attacks proposed in prior research using modern hardware virtualization extensions and evaluated their pertinence on open-source hypervisors. We further discussed inherent limitations in using hardware virtualization extensions for VMM-based security monitoring and highlighted how these limitations can have unintended side effects when not taken into consideration. We concluded with exploring Intel's system-management mode and discussed how it can be used to both implement and to subvert VMI applications.

Acknowledgments

The research leading to these results was supported by the Bavarian State Ministry of Education, Science and the Arts as part of the FORSEC research association.

6. REFERENCES

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, October 2012.

[2] ARM. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, July 2012.

[3] 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.* IEEE, 2010.

[4] R. R. Collins. The caveats of system management mode. `http://www.rcollins.org/ddj/May97/May97.html`, October 29 2014.

[5] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, New York, NY, USA, 2013. ACM.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security.* ACM, 2008.

[7] B. Dolan-Gavitt, B. Payne, and W. Lee. Leveraging forensic tools for virtual machine introspection. Gt-cs-11-05, Georgia Institute of Technology, 2011.

[1] MSR_IA32_VMX_BASIC_INFO_VMCS_DUAL_MON

[8] N. Elhage. Virtunoid: Breaking out of kvm. *Black Hat USA*, 2011.

[9] GRSecurity. Pageexec. `https://pax.grsecurity.net/docs/pageexec.txt`, December 30 2006.

[10] T. Haruyama and H. Suzuki. One-byte modifications for breaking memory forensic analysis. *Black Hat Europe*, 2012.

[11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3*, June 2013.

[12] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. Sok: Introspections on trust and the semantic gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 605–620, Washington, DC, USA, 2014. IEEE Computer Society.

[13] A. Lagar-Cavilla and A. Cooper. Xen-devel: Handle resumed instruction based on previous mem_event reply. `http://www.gossamer-threads.com/lists/xen/devel/347492#347492`, September 11 2014.

[14] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*.

[15] P. Markowsky. Ring -1 vs. ring -2: Containerizing malicious smm interrupt handlers on amd-v. *ShmooCon*, 2010.

[16] R. Riley, X. Jiang, and D. Xu. An architectural approach to preventing code injection attacks. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):351–365, 2010.

[17] J. Rutkowska and R. Wojtczuk. Preventing and detecting xen hypervisor subversions. *Blackhat Briefings USA*, 2008.

[18] S. Sparks and J. Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, pages 504–533, 2005.

[19] S. Sparks and S. Embleton. Smm rootkits: A new breed of os independent malware. *Black Hat USA, Las Vegas, NV, USA*, 2008.

[20] J. Torrey. More: measurement of running executables. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, pages 117–120. ACM, 2014.

[21] J. Torrey. More shadow walker: Tlb-splitting on modern x86. BlackHat, 2014.

[22] K. Tseng, B. Liu, R. Sood, M. Castelino, and M. Tallam. Determining policy actions for the handling of data read/write extended page table violations, June 27 2013. WO Patent App. PCT/US2011/067,038.

[23] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[24] J. Wang, K. Sun, and A. Stavrou. An analysis of system management mode (smm)-based integrity checking systems and evasion attacks. *George Mason University Department of Computer Science Technical Report*, 2011.

[25] F. Wecherowski. A real smm rootkit: Reversing and hooking bios smi handlers. *Phrack Magazine*, 13(66), 2009.

[26] C. Willems, R. Hund, and T. Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. Technical report, Ruhr-Universitat Bochum, 2013.

[27] R. Wojtczuk and J. Rutkowska. Attacking smm memory via intel cpu cache poisoning. *Invisible Things Lab*, 2009.

[28] G. Wurster, P. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pages 127–138. IEEE, 2005.

[29] F. Zhang, K. Leach, K. Sun, and A. Stavrou. Spectre: A dependable introspection framework via system management mode. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.