Hypervisor based Memory Introspection: Challenges, Problems and Limitations

Andrei Lutas[1], Daniel Ticle[1] and Octavian Cret[2]
[1]*Bitdefender, 1 Cuza Voda Str., City Business Center, Bitdefender, 400107, Cluj-Napoca, Romania*
[2]*Computer Science Department, Technical University of Cluj-Napoca,*
26-28 Gh. Baritiu Str., 400027, Cluj-Napoca, Cluj, Romania
{*vlutas, dticle*}*@bitdefender.com, octavian.cret@cs.utcluj.ro*

Keywords: Hypervisor, Introspection, Challenges, Limitations, Solutions.

Abstract: Hypervisor-based memory introspection is a well-known topic, in both academia and the industry. It is accepted that this technique brings great advantages from a security perspective, but it is known, as well, that this comes at greater implementation complexity and performance penalty. While the most obvious challenges, such as the semantic gap, have been greatly discussed in the literature, we aim to elaborate on the engineering and implementation challenges encountered while developing a hypervisor-based memory introspection solution and to offer theoretical and practical solutions for them.

1 INTRODUCTION

Traditionally, one thinks about security as a piece of software that runs inside the operating system (OS), providing services such as file scanning or application monitoring. Due to the increasing complexity of malware and attacks, security solutions had to rely more on isolation, thus certain components were moved from the user space into the kernel space, making attacks more challenging. Lately, an increasing number of attacks rely on complex techniques such as exploits and privilege escalation that can easily render a security solution inert. Thanks to the latest advances in hardware virtualization, designers can now take advantage of features such as hardware-enforced isolation and use several extensions in order to provide increased security in a hardware-isolated environment.

Garfinkel and Rosennblum first proposed the memory introspection technique in 2003 (Garfinkel and Rosenblum, 2003). It involves moving the security solution outside the OS, thus isolating it from possible attacks from within the virtual machine. The main challenge of this technique, the semantic gap, was thoroughly discussed in papers such as (Carbone et al., 2009), (Baliga et al., 2008), (Cozzie et al., 2008), (Dolan-Gavitt et al., 2009) or (Lin et al., 2011), but creating an HVI (*hypervisor-based memory introspection*) solution involves some low-level engineering challenges that have not been discussed in detail so far.

In this paper, we aim at detailing on engineering challenges encountered while developing a real-time hypervisor-based memory introspection engine. This is a very serious engineering challenge that requires complex knowledge from many domains: OS, computer security, low-level programming, etc. We have personally faced these challenges while developing U-HIPE (Lutas et al., 2015a): a hypervisor-based memory introspection engine, capable of protecting both the kernel space of the OS and the user space of the applications. The next section contains a brief introduction in the virtualization fields. The third section details how memory introspection works, while the fourth section will detail on some of the most important engineering challenges encountered. The conclusions are drawn in section five.

2 HARDWARE VIRTUALIZATION A BRIEF OVERVIEW

Hardware-based virtualization has first been introduced in 1960, in the experimental IBM M44/44X. In 2005, it was introduced in x86 CPUs, as the SVM (*Secured Virtual Machine*) extensions on AMD (AMD Corporation, 2005) and VT-x (*Virtualization Extensions*) on Intel (Intel Corporation, 2016b). The key role of virtualization is to allow multiple guest OSs, or virtual machines (named from here on *VMs* or *guests*)

to run concurrently on a host system. A virtual-machine monitor, named from here on *VMM* or *hypervisor*, controls all these VMs. A hypervisor generally uses a trap & emulate architecture, where the CPU generates an event (*VM exit*) whenever it needs special handling from the hypervisor. Examples of VM exits include executing a privileged instruction, accessing restricted I/O ports or MSRs, an external interrupt or accessing restricted memory pages. After finishing handling the event, the hypervisor returns the control to the interrupted guest via a *VM entry*.

The core structure of any hypervisor is the VMCS on Intel (*Virtual-Machine Control Structure*) or VMCB (*Virtual Machine Control Block*) on AMD (named from here on VMCS) which represents a virtual CPU (or *VCPU*). The VMCS contains all the essential information about the VCPUs: the host state, the guest state, the guest control area, VM exit and entry control and VM exit information. This structure contains the saved state of the guest or hypervisor on VM entries and VM exits, and control fields that configure how the CPU should handle various events and instructions.

VT-x and SVM were further extended with memory virtualization capabilities a second level address translation, that allows the hypervisor to directly configure a mapping from guest-physical addresses to host-physical addresses. The second level address translation (SLAT), named EPT on Intel (*Extended Page Tables*) and NPT (*Nested Page Tables*) on AMD, has a structure similar to that of the legacy IA page tables, with entries containing control bits that configure read, write or execute access. A VM has its own SLAT structure just like a regular process has its own page table hierarchy. The SLAT is fully controlled by the hypervisor, and thus enables it to enforce page-level access restrictions over the guest-physical memory, without interfering with the guest page tables. When SLAT is in use, there are three different types of addresses on a host system: guest-virtual addresses (1) are those addresses normally used by programs inside the VM; these translate via the legacy IA page tables into guest-physical addresses (2), which are further translated into host-physical addresses (3) using the SLAT, which are then accessed by the hardware. Figure 1 shows the memory translation mechanisms in a virtualized system.

3 VIRTUAL MACHINE INTROSPECTION

Hypervisor-based VM introspection is a technique of analysing the state and behaviour of a VM from the

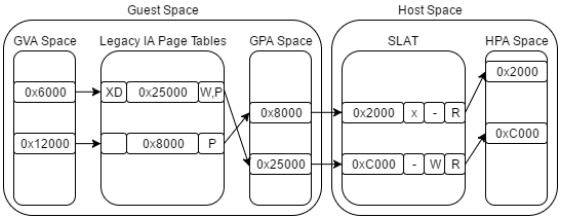

Figure 1: Address translation with SLAT active.

outside (from hypervisor's level). It involves accessing the hardware state (CPU registers) and the physical memory of the analysed VM. However, this information is not sufficient one needs to correlate this low-level data with OS specific structures and events in order to gain knowledge about the VM state. This process is known as *bridging the semantic gap*, and several solutions were proposed for it, as mentioned in section 1. Once meaningful structures have been identified inside the guest VM, the hypervisor could configure, using the SLAT, restricted access for certain structures. In general, the process of protecting memory sits at the heart of hypervisor based introspection, and the challenges that are discussed here refer mainly to it. In addition, the VM introspection can be done both on-premise (for instance, memory analysis of a live VM or a dump for forensics) and in real-time (where the guest behaviour is continuously monitored and where performance is critical). In this paper, we will focus on the second approach.

By using the SLAT, the introspection solution can enforce restricted access to various areas of the memory. For example, kernel code-pages could be marked as being non-writeable, thus preventing rootkits (advanced kernel malware) (G. Hoglund and J. Butler, 2005) from placing inline hooks inside them. On the other hand, the introspection solution may enforce no-execute policy on certain areas such as the stacks or heaps, thus intercepting any attempt to execute code from those areas, which are almost always indicative of an attack. No-read policies can also be used, for example, to hide code or data that the introspection solution protects inside the guest (for example, from an in-guest security solution). The normal flow of events on a protected VM looks like the one depicted in Figure 2. There are two main possibilities for handling each event intercepted by the introspection logic: it will either be emulated if it is considered legitimate or blocked, if it is deemed to be malicious. While blocking an attack is simple (the faulting instruction can simply be skipped), legitimate accesses must be emulated. In general, there are two types of such cases:

1. Benign accesses inside protected structures. In some cases, the introspection logic wishes to allow certain components (usually belonging to the

OS itself) to access the protected structure;

2. If the protected structure is less than 4K (minimum page size), and since the protection works by restricting access for 4K guest-physical pages inside the SLAT, there may be cases where the software accesses other structures located inside the same page with the protected structure;

Creating a security solution than runs outside the protected VM is a very complex task, but there are several reasons why it is worth implementing:

1. *Hardware-enforced isolation*. A traditional security solution runs within the VM, and is susceptible to attacks: even if it runs in the most privileged mode inside the OS kernel, an attack may employ complex techniques such as privilege escalation (ref, a) in order to gain higher privileges. When the malware runs at the same privilege level as the security solution, it can easily bypass or disable it.
2. *Ability to monitor CPU-level events*. Certain events cannot be monitored from within the VM. For example, it is not possible to be notified when a hardware register is being modified. When running inside a hypervisor, there are multiple CPU-level events that the introspection logic can intercept in order to provide security; for instance, the system call registers may be modified by rootkits in order to place a system-wide syscall hook.
3. *Ability to monitor memory without interfeering with the OS*. It is not possible to monitor memory accesses inside the VM without heavily interfering with the OS. When running inside a hypervisor, one can leverage the SLAT in order to place restrictions on guest-physical pages, beyond the OS capabilities.
4. *Increased usage of virtualized environments*. Cloud service providers heavily rely on virtualization, which offers the capability of securing multiple VMs without having to install a security solution in each one of them.

It is therefore clear that hypervisor-based introspection is a solution to current security, scalability and deployability demands. In addition, CPU vendors keep adding new features that can help improve performance for such an application: virtualization exception, VM functions and probably more are yet to be revealed (D. Durham, 2014).

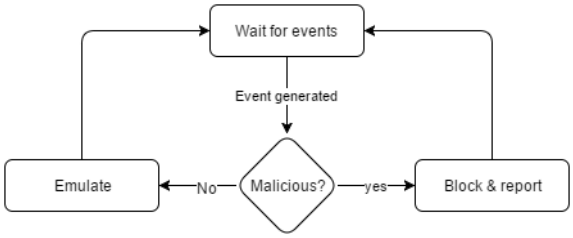

Figure 2: Introspection events handling flow.

4 CHALLENGES AND SOLUTIONS

As discussed earlier, creating a hypervisor-based memory introspection solution has significant advantages: isolation, greater visibility inside the monitored system and the possibility to provide CPU and memory protection that would otherwise be very difficult to achieve. Aside the obvious challenge of the semantic gap, there are several other low-level engineering challenges that arise when developing such a solution.

4.1 Accessing the Guest Memory

Challenges. As already detailed in the previous section, the SLAT mechanism introduces a new level of address translation and has to deal with accessing guest-virtual and guest-physical memory. In both cases, the translation of the accessed memory must be handled by the introspection logic (Lutas et al., 2015b), in order to ensure that the page is present and to obtain the final host-physical address.

Mapping and translation events can described by two functions: f for mappings and g for translations. We first define the function f_{hpa} that maps host-physical memory. This function needs to find a free region inside the hypervisor virtual address space and add a new entry for it, thus making it accessible. Translation Lookaside Buffer (TLB) invalidations on all the physical CPUs would further be required, either when mapping or unmapping the desired page. Once such a function is available, we need a function f_{gpa} that maps guest-physical memory. In order to do that, we also need a function g_{gpa} that translates the given guest-physical address into a host-physical address using the SLAT tables. This involves walking each level of translation and validating the results at each step. Finally, in order to map guest-virtual memory, we need f_{gpa} together with a new function, g_{gva} capable of translating the guest-virtual address into a guest-physical address, using the guest legacy IA page tables. If we monitor a guest OS running in long mode, the legacy IA page tables will be four levels deep, meaning that mapping one guest-virtual page v would lead to the following sequence of function calls:

$$f_{gva}(v) = f_{gpa}(g_{gva}(v)) \quad (1)$$

Each translation involves more mappings: translating the guest-virtual address v into a guest-physical address p involves 4 physical address mappings, one for each level of translation (page map level 4, page directory pointer table, page directory and page table) assuming long-mode paging:

$$g_{gva}(v) = f_{gpa}(pml4) + f_{gpa}(pdp) + f_{gpa}(pd) + f_{gpa}(pt) \quad (2)$$

The f_{gpa} function implies two steps as well:

$$f_{gpa}(p) = f_{hpa}(g_{gpa}(p)) \quad (3)$$

The final sequence of function calls is complex, involving mapping five different guest-physical pages, which in turn translate to mapping five different host-physical pages. Such events may be rare in some scenarios, but when a high-performance, real-time, user-mode memory introspection solution is desired, the events may be dense enough to pose performance issues. Figure 3 shows the time spent, on average, inside memory mapping routines on a 64 bit Windows 8.1 system, in each event, in a normal usage scenario. It is worth mentioning that some intervals include each other: for example, the Map GVA interval includes portions of the Translate GVA and Map GPA intervals, since mapping a guest-virtual page involve multiple physical mappings.

Solutions. There are several optimization options:

1. Keeping the entire host-physical space mapped inside the host virtual space. This eliminates the need to map host-physical memory, as it would be already mapped at a predetermined address. This solution considerably improves the performance, but it has the drawback of using a significant portion of the host virtual address space to map the guest space. Furthermore, while modern CPUs use 48 bits of virtual addresses (in long mode), the width of the physical addresses may exceed 40 bits (it is currently defined to be up to 52 bits), thus making this approach unscalable.

2. Using caches, as a trade-off between scalability and keeping pages mapped inside the introspection memory space. According to our tests, it is clear that the improvement is significant (see Figure 4). But caches increase the code-base of the introspection solution and may be complex to implement, as they need to be flushed on certain events (for example, translation modifications). The caches we implemented simply maintain a batch of often-accessed pages mapped inside the introspection solution's address space, thus avoiding costly translations and memory mappings.

3. Dedicated CPU instructions. This would be the most efficient solution, bringing native performance for guest memory access. Some research in this direction has already been done (Serebrin and Haertel, 2008). The CPU could implement instructions capable of reading guest-physical or guest-virtual memory. The access would be done in the context of the current VMCS, thus directly exposing both the guest CR3 and the SLAT tables used by that VM. We cannot accurately estimate the complexity of such a solution, but it should offer native performance, similar to regular memory accesses (Lutas et al., 2015b).

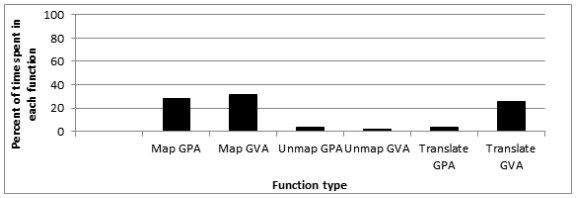

Figure 3: Percent of time spent in memory-mapping routines.

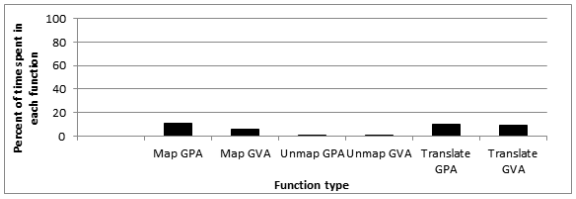

Figure 4: Percent of time spent in memory-mapping routines (with caches).

4.2 Protecting Pageable Memory

Challenges. Protecting memory inside SLAT works only for guest-physical pages; however, the OS and the applications use guest-virtual addresses, which translate to guest-physical addresses. In order to provide protection for any given guest-virtual page, the introspection solution has to first translate it into a guest-physical page, by using the in-guest legacy IA page tables. This strategy is effective as long as the translation does not change; if it does, the introspection solution must find a way to maintain protection on the guest-virtual address. There are three cases that must be handled (Lutas et al., 2015a):

1. Page is removed from memory (swapped-out);
2. Page is brought in memory (swapped-in);
3. Page is moved somewhere else.

Another problem may be maintaining page table protection among various virtual address spaces: each process has its own, private virtual address space, represented by a dedicated page table hierarchy. While

modern OSs, such as Windows and Linux share the kernel space amongst all the processes (meaning that the kernel space is global and identical in every virtual address space), an attacker may build custom page table hierarchies that would lead to malicious translations. In order to avoid this, introspection logic would have to intercept CR3 loads and make sure it intercepts the page tables in every existing virtual address space. This has a significant negative impact on the performance: the number of CR3 loads will increase linearly with the number of existing processes.

Solutions. If not handled properly, these situations may lead to undesired effects, such as loosing protection on the page or protecting an undesired page. It is imperative to find a way to intercept such swap events. The most obvious way to do this is by intercepting the guest page tables. The implementation of this method is complex and it leads to significant performance penalties, as for every guest-virtual page, one needs to intercept writes in up to 5 actual guest-physical pages: one for the actual guest-physical page that is translated from the protected page and up to four more writes on each page table that translate the given guest-virtual address. By intercepting the entire hierarchy of page tables, the introspection logic ensures that any translation modification would be trapped via an EPT violation, thus allowing the protection to be adjusted. On each page table write, the introspection logic needs to decode the written value, in order to analyse the modification type. While the OS may modify several bits inside the entries (for example, the accessed or dirty flags), only three types of events are of particular interest:

1. The new value has the present bit set, while the old value has the present bit 0: this is a swap-in operation, meaning that the guest-virtual page has just been mapped back into the memory; the introspection logic must add protection on the newly mapped guest-physical page;
2. The new entry has the present bit 0, and the old entry has the present bit set; this means that the page is being swapped out, and the introspection logic has to unprotect the underlying guest-physical page;
3. Both the new and the old entry have the present bit set, but the guest-physical addresses are different; in this case, the introspection logic must unprotect the old page and add protection on the new page.

4.3 Accessing Swapped-out Memory

Challenges. Inside the kernel mode, most of the critical data structures are always present in physical memory they are non-paged, which means that the OS will never swap them out. Doing kernel introspection is, therefore, usually very straightforward. In the case of process memory introspection, it is usually the reverse only the most accessed pages are committed and present inside the physical memory, while all the other pages are swapped out. The introspection logic may need to access such swapped-out pages in order to properly identify the process-specific structures, but it cannot do so if these pages are not present inside the physical memory.

Solutions. Here are some ways to solve this issue:

1. Directly access the swapped out data inside the swap file. This can be very difficult, since the format of the swap file is highly specific to the OS and the introspection logic would need access to the storage device where the swap file resides;
2. Intercept writes inside the IA page tables entries that translate the needed page and wait for it to be swapped in. This has the advantage of simplicity, as it only relies on permission modifications inside SLAT, but it does not ensure that the page will ever be swapped in;
3. Forcefully inject a page-fault exception inside the guest (Lutas et al., 2015a). When employing the technique from point 2, one can force the needed page to be swapped in by injecting a page-fault inside the guest. This adds complexity to the previous technique, as the page-fault must be injected with care; the correct process context must be loaded and the OS must accept the fault. There are cases, for example high IRQL (M. Rusinovich and D. Solomon and A. Ionescu, 2012) on Windows, where a page-fault would cause on OS crash. While in user-mode, the OS will always accept page-faults. This method has been already described in detail in (Lutas et al., 2015a).

4.4 4K Granularity

Challenges. As previously mentioned, one of the key roles of hypervisor-based memory introspection is memory monitoring. This can be done by using the SLAT, as it is fully in the hypervisors control. One way to provide protection and monitor the guest VM is by modifying guest-physical pages permissions inside the SLAT. For example, one could intercept all write operations inside a given guest-physical page by setting the *write* bit to 0. This way, any instruction or event that would cause a memory store inside that guest-physical page would trigger a SLAT violation. The memory introspection logic can analyse the event and decide whether it is legitimate or not.

The problem is that the minimum granularity is the minimum size of an EPT page (4K). In order to intercept writes inside a smaller range, one has to handle writes outside the given range, as well. While there are multiple ways to address this issue, the number of writes inside the 4K page, but outside the protected range, might induce a significant performance impact. Depending on the specific of the introspection logic, this may severely limit either the functionality or the performance of the solution. A simple example is the protection of small, heap or pool allocated objects, such as processes or threads. These structures are usually smaller than a single page, and even inside these structures, one may not wish to intercept writes on every field. A security application may protect, for example, a field that links the structure into a global list of such objects, or a field that contains security related information, in order to prevent malicious modifications. In this case, a word size or a small multiple of a word size interception would be useful. However, protecting such a small range inside the page may incur a significant performance penalty, as the same page can contain multiple other structures allocated in it. Even in the same protected structure, there may be portions that are written very often. In many cases, the performance impact of intercepting a small range of any structure inside a page of memory will be difficult to predict: it may be near zero if that page does not contain any other allocated structures or it may be extremely high if volatile read-write structures are allocated there.

Solutions. Here is a list of possible solutions:

1. Make sure each protected structure is allocated at a page boundary. This approach has 3 drawbacks:
   - The memory allocator must be intercepted;
   - Writes inside the structure can still take place;
   - Increased memory usage.

   This solution can be a compromise between complexity and performance. However, it may not be very scalable, as forcing many structures to be allocated into their own page can cause serious memory consumption. For example, 100 128B sized structures would cause a total of almost 400KB of space to be wasted. In addition, simply intercepting the memory allocator may induce additional performance overhead, although deterministic and not noticeable, according to our tests (Lutas et al., 2015a).
2. CPU extensions. Ideally, all the unwanted exits would be eliminated if the CPU would provide a bit-level protection mechanism. This is probably extremely difficult to achieve in today's hardware, but a granularity of several bytes (Sahita et al., 2014) (for example, 128 bytes) would be more than enough to offer significant performance gain for existing solutions and to offer the possibility to protect new structures. We have used the 128B (2 x cache line size) granularity and we have measured the potential performance improvement for various data structures, on Windows; the results are illustrated in Figure 5

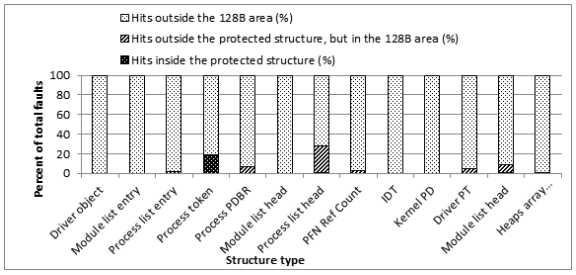

Figure 5: 128 bytes granularity protection stats.

4.5 Accessed and Dirty Bits

Challenges. Whenever the CPU does a page walk, it sets the accessed and dirty bits (A/D bits) inside the page tables. These writes trigger a SLAT violation if the page tables are marked as non-writable. Handling these events by the introspection logic can be done either by single stepping the instruction that triggered it, or by emulating the entire page walk. The A/D bits may not be of a significant importance for the introspection logic: in our particular implementation, we ignore them entirely. In order to asses the performance impact induced by the page-walker, we have simulated a high memory pressure scenario, where large amounts of memory are allocated, used and freed frequently. We ran the simulation for 10 minutes, on Windows 8.1 x64, and we plotted the number and percentage of A/D bits induced EPT violation in each second, creating two histograms: one with the absolute number of A/D bit induced faults and one with the fraction of A/D bit induced faults out of the total number of EPT violations. The results can be seen in Figure 6 and Figure 7. In figure 7, one can notice that often the number of page-walker caused faults exceeds 100,000 per second and many times the totality of the EPT violations are represented by such events.

Solutions. Unlike the other challenges discussed so far, this one is difficult to avoid directly in software. Here are some of the possible solutions:

1. *Always keep A/D bits set*. This ensures that once the CPU sets the A/D bits inside a page table, the OS will never reset them. This has the disadvantage of being invasive and it may even cause disruptions to the OS memory manager.

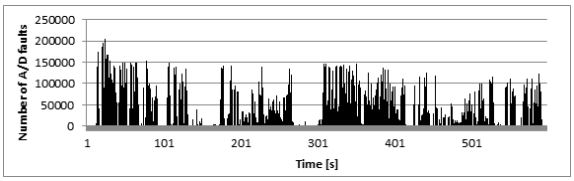

Figure 6: Frequency of A/D bit induced EPT violations in a 10 minutes time frame.

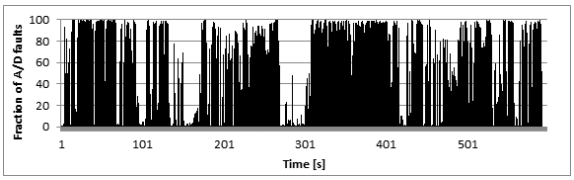

Figure 7: Fraction of A/D bit induced faults in a 10 minutes time frame.

2. *Hardware Solution.* Some CPUs (specifically Intel) already provide a dedicated bit inside a VMCS field that indicates when an EPT violation was caused by the CPU page walker (AMD Corporation, 2005, Vol. 3, Ch. 27). It may be possible to avoid these exits altogether, and simply ignore the page walker whenever it accesses the page tables. Just like the other hardware extensions, this is also speculative with regard to the complexity of the possible implementation, but the improvement would be significant: not only would the A/D bit exits be removed, but a page-walker would no longer be needed inside the introspection logic.

4.6 Instruction Decoding

Challenges. When dealing with SLAT faults, the introspection logic must decide whether the access is legitimate or not. Many times, in order to do so, the instruction that caused the fault must be analysed, in order to obtain some information such as the size of the access or the new value stored in memory, so an instruction decoder must be part of the introspection logic. While simply decoding an instruction in order to determine the operands and access size is not a very computational intensive task, there are some special cases and events, due to the CISC character of the x86 instructions set, that need special handling and cannot be generically treated:

1. IDT (Interrupt Descriptor Table) accesses as part of an exception or interrupt delivery. There is no dedicated field inside the VMCS to indicate this kind of access, and intercepting reads inside the page that contains the IDT may lead to such events. This kind of event could be handled by checking dedicated VMCS fields indicative of event delivery, which means that the fault took place as a result of such an event;
2. GDT (Global Descriptor Table) or LDT (Local Descriptor Table) accesses as part of loading a segment descriptor or setting an accessed bit inside a segment descriptor. This kind of events can be handled by inspecting the faulting instruction - any instruction that loads a segment, for example, will also load the underlying descriptor;
3. TSS (Task State Segment) accesses as part of a task switch or delivery of an interrupt or exception. Certain branch instructions may lead to a task switch in certain conditions; this can be inferred from the instruction itself. A more tricky case is a task switch that takes place as a result of an exception or interrupt;
4. BTS (Branch Trace Store) and PEBS (Precise Event Based Sampling) memory stores. These take place asynchronously and decoding the instruction at which boundary the fault took place will not be helpful. A possible solution is to check the memory range of the faulted address (which can be done by inspecting certain MSRs), and seeing if it lies within BTS or PEBS region;
5. PT (Processor Trace) stores. These are very similar with the BTS/PEBS events, and they may be detected by checking the faulted address against the PT memory range.

These events, although peculiar with respect to SLAT faults, may be expected and properly detected if the instruction is carefully inspected or if additional validations are made, but in some cases, it is very difficult to determine the size of such a memory access. In the case of IDT, GDT, LDT or TSS, the size of the access would be obvious (one or two memory words); in case of a BTS, PEBS or PT store the size may not be directly obtainable, leading to a possible problem in handling the fault event. In addition to these events, there are also instructions that need special handling:

1. XSAVE/XRSTOR instructions. These instructions operate on multiple register sets and the access size cannot be directly determined. Instead, an introspection logic has to query for enabled features in XCR0 (Extended Control Register 0) or IA32_XSS_ENABLE. In addition, it has to do several CPUID queries to determine exactly the offsets inside de XSAVE area and the size of each saved component;
2. MPX instructions. BNDLDX and BNDSTX have the documented side-effect of doing a load or a store inside the bounds tables. This structure is somewhat similar to the IA page tables, and it

is not encoded in the instruction its base can be extracted from the BNDCFGU (in user-mode) or BNDCFGS (in kernel-mode) register. It is two levels deep, so special care must be taken when checking for bound access, in order to properly handle both bound directory and bound table accesses;

3. CET enabled instructions. If Intel Control flow Enforcement Technology is enabled (Intel Corporation, 2016a), the behaviour of some instructions changes. The most important modification appears at the level of procedure call and return instructions: these access a new CPU structure, called the *Shadow Stack*. Procedure calls automatically store the return address on this new structure, and return instructions load it, in order to make sure it hasn't been altered. The shadow stack is pointed by a new CPU register, called the SSP (*Shadow Stack Pointer*), and this capability can be enabled for both user and supervisor code. Initial documentation doesn't indicate whether a shadow stack access is flagged in the VMCS, so specific range checks can be made, in order to see if the accessed address lies within that range;
4. Scatter-gather instructions. This is a special class of instructions introduced in the AVX2 and AVX512F instruction sets that are capable of accessing multiple separate memory addresses - these addresses can even reside in separate pages. For example, the instruction *VPSCATTERDD [RAX + ZMM0], ZMM1* can write to up to 16 different memory locations;
5. String instructions. Instructions such as MOVS both read and write memory, so special handling is needed, since they cause both read and write SLAT faults;
6. Instructions that may trigger another type of VM exit before doing the memory access. These instructions will trigger a VM exit before actual execution (before accessing the memory and getting a change to cause a SLAT fault). The introspection logic needs to make sure that on every event pertaining the execution of such an instruction it will also validate the accessed guest-virtual address and guest-physical address against the legacy IA page tables and SLAT permissions. For example, the guest VM may attempt to execute XSAVES instruction, which is configured to cause a VM exit. The hypervisor may emulate the instruction without fully validating the SLAT, and thus it may bypass protections established by the introspection logic. Instructions that can cause a specific VM exit and that can also access memory include string I/O instructions, VMX instructions or descriptor table accesses.

Solution. Although there are many special cases that must be handled by the introspection logic, usually there are a small number of instructions that trigger events. In our tests, we have discovered that more than 95% of all the instructions that ever trigger an EPT violation are simple MOV instruction. This behaviour is constant on both 32 and 64-bit OSs and has been confirmed on both Windows and Linux.

In addition to the logic handling challenges, instruction decoding constitutes another problem. While this can be done quite easily, and there are several disassemblers available (i.e. Capstone, distorm, udis86, etc.), the introspection logic has to repeat this task on every VM exit. Mapping memory and decoding the instruction on each exit can be very expensive, so an instruction cache can be used: save tuples (instruction pointer, decoded instruction) on each event, which allows a fast lookup of the faulting instruction on future events. If a new VM exit is triggered from an instruction pointer that has already been cached, the introspection logic can retrieve the decoded instruction directly from there, thus significantly improving performance. In addition, the CPU might provide basic information to the hypervisor, such as the access size that caused the fault or the instruction bytes, thus relieving the hypervisor or the introspection logic from doing costly decodes.

4.7 Instruction Emulation

Challenge. Legitimate memory accesses that trigger SLAT faults must be handled by the introspection logic. There are two possible *solutions* to this:

1. *Instruction Emulation.* An instruction emulator must simulate the behaviour of the emulated instruction entirely. Not only is this complex, but it should also be able to handle every instruction supported by the target ISA. In the case of the x86 architecture, the ISA has been greatly extended over the years and it now contains several thousand different instructions, some of which are very complex to handle. The code base of a complete emulator would be of significant size, therefore increasing the attack surface and making the introspection logic or the hypervisor vulnerable (ref, b). In addition, an emulator would have to pay extreme attention to special cases such as cross-page accesses, time of check vs. time of use or instructions that access multiple pages and are capable of triggering more that one SLAT fault;
2. Single stepping. This approach has the great advantage of being generic: it can handle any

instruction without needing special knowledge about it. In addition, this mechanism does not have to specially handle instructions that cause multiple faults: its incremental nature (granting permission for each page once it is validated) ensures that an arbitrary number of concomitant faults can be handled. This mechanism works by temporary removing the protection from the accessed page, thus allowing the instruction to complete successfully. If the instruction would trigger a new fault, the mechanism can be invoked in a recursive manner. While the protection is disabled, the introspection logic has to make sure that other VCPUs won't modify the content of the pages while the single-stepping occurs. This can be done in at least two ways:

- Pause all the other VCPUs while the faulting one single-steps the instruction. This ensures that only the faulting VCPU runs code while the protection is removed from SLAT;
- Create a dedicated single-step SLAT that would be used only while single-stepping the faulting instruction. This allows use to load a new SLAT on the VCPU that will single-step the access.

In each case, a significant performance impact would accompany this technique: at least two separate VM exits would have to be handled for each single-stepped instruction - one for the actual event that triggered the single-stepping and one more when finishing single-stepping, that would allow us to resume all other VCPUs or restore the original SLAT. A better approach would be to use an instruction emulator for the most common instructions (for example, the MOV instruction, which accounts for more than 90% of all the VM exits) and use the single-step mechanism only for unsupported instructions, thus using the great advantages from both techniques. A security issue with both emulation and single-stepping is the cross-VCPU instruction modification attack (ref, b). Such an attack would leverage multi-processors systems in order to modify an instruction at the right moment: after it has generated a VM exit and has been analysed, but before being emulated or single-stepped. Such an attack can be prevented by properly validating the instruction before emulation or single-stepping.

Since single-stepping is the most desired approach, the hardware could help into achieving this more efficiently by allowing the hypervisor to temporary override SLAT permissions. Such a mechanism would involve one or more fields in the VMCS that would contain a guest-physical address together with the override bits. While enabled, such a mechanism would allow the faulting instruction to access a given address if it is in the override list, even if SLAT would otherwise deny it. The single-stepping mechanism could be automatically disabled by the CPU once the faulting instruction has been executed. Therefore, we would not need an emulator or a specific single-stepping mechanism, and one could handle any instruction by simply writing to some fields inside the VMCS. In order to handle instructions that cause multiple faults, several such guest-physical override fields could be present. In our tests, we discovered that in more than 57% of the cases, the instruction can be single-stepped in a single iteration (the instruction triggers only one SLAT fault), and about 20% of the instructions trigger two and three faults. The detailed results can be seen in Figure 8. Less than 1% of the instructions trigger 5, 6 or 7 faults, and we did not encounter any instruction that triggers 8 or more faults.

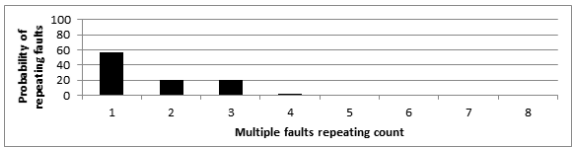

Figure 8: Probability of multiple SLAT-faults events.

The reason why multiple faults can be triggered by a single instruction is that multiple guest-physical pages are accessed when executing any given instruction. For instance (ignoring the accesses made to fetch the instruction), an instruction that writes a single byte inside a given page may cause five different guest-physical accesses in long mode: four accesses inside the IA page tables and one access inside the actual page. An instruction such as MOVS that makes a page-boundary access may access 2 pages when reading, 2 pages when writing and up to 16 page tables (2 x 4 + 2 x 4). These cases are rare, however, and if they occur, they could be handled directly in software.

5 CONCLUSIONS

We presented in this paper low-level engineering challenges that we have encountered during the development of a hypervisor-based memory introspection engine. While we do not claim that this is a complete list or that the solutions that we implemented or proposed are the only ones possible, we think they are representative, relatively easy to implement and effective.

The most problematic challenges are the hardware limitations that also induce the highest performance impact: page granularity protection and accessed/dirty bits. While we cannot asses the complexity or costs of implementing the proposed extensions

in hardware, we strongly believe we will see them in the future assisting memory introspection solutions.

Other challenges relate to instruction decoding and emulation, and while they can be handled in software using caches and emulators, they involve deep knowledge of the instruction set and behaviour of the CPU. Problems such as protecting paged memory or accessing swapped out pages may not appear in a kernel-mode memory introspection scenario, but are very common when dealing with user-mode memory introspection. While the solutions are not necessary complex, they are neither obvious nor straightforward to implement and may not be very effective.

We have discussed various possible improvements that could be made inside the CPU itself in order to aid memory introspection tasks, and while they are purely theoretical, they may bring significant improvement to such applications, both from the implementation complexity and performance perspective. The complexity of implementing these in the CPU, however, may vary significantly, although emulators such as Bochs or QEMU and simulation tools such as PIN may provide an overview on how such extensions may improve memory introspection. Many hardware extensions were implemented recently for various algorithms, like AES, SHA or CRC, showing an obvious trend of moving as much logic as possible on the chip.

The software improvements that we have discussed were implemented and tested in U-HIPE and some of them were presented in papers such as (Lutas et al., 2015a) and (Lutas et al., 2015b), and, while the performance increases, so do the attack surface and the implementation complexity.

It is worth mentioning that currently, introspection solutions are somewhat ahead of their time: they are complex software that leverage the latest CPU innovations in order to provide security, although the vast majority of these extensions were not created for this specific purpose. We keep seeing significant improvements in hardware, especially in security & virtualization fields, and we think that future CPU generations will include extensions that may help fix at least some of these issues, making hypervisor-based memory introspection solutions easier the develop, deploy and much more efficient.

REFERENCES

AMD Corporation (2005). *AMD64 Virtualization Codenamed Pacifica Technology. Secure Virtual Machine Architecture Reference Manual.*

Baliga, A., Ganapathy, V., and Iftode, L. (2008). Automatic Inference and Enforcement of Kernel Data Structure Invariants. In *In Proc. Annual Computer Security Applications Conference*, pages 77–86.

Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. (2009). Mapping kernelobjects to enable systematic integrity checking. In *In Proc. The 16th ACM conference on Computer and communications security Pages*, pages 555–565.

Cozzie, A., Stratton, F., Xue, H., and King, S. T. (2008). Digging for data structures. In *In Proc. 8th USENIX conference on Operating systems design and implementation*, pages 255–266.

D. Durham (2014). Mitigating Exploits, Rootkits and Advanced Persistent Threats.

Dolan-Gavitt, B., Srivastava, A., Traynor, P., and Giffin, J. (2009). Robust signatures for kernel data structures. In *In Proc. 16th ACM conference on Computer and communications security*, pages 566–577.

G. Hoglund and J. Butler (2005). *Rootkits: Subverting the Windows Kernel*.

Garfinkel, T. and Rosenblum, M. (2003). A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206.

Intel Corporation (2016a). *Control-flow Enforcement Technology Preview*.

Intel Corporation (2016b). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-060US.

Lin, Z., Rhee, J., Zhang, X., Xu, D., and Jiang, X. (2011). Graph-based signatures for kernel data structures. In *In Proc. 12th Annual Information Security Symposium*, page Article no. 21.

Lutas, A., Colesa, A., Lukacs, S., and Lutas, D. (2015a). U-HIPE: hypervisor-based protection of user-mode processes in Windows.

Lutas, A., Lukacs, S., Colesa, A., and Lutas, D. (2015b). Proposed Processor Extensions for Signicant Speedup of Hypervisor Memory Introspection. In *Trust and Trustworthy Computing*, pages 249–267.

M. Rusinovich and D. Solomon and A. Ionescu (2012). *Windows Internals 6th edition*.

Sahita, R., Shanbhogue, V., Neiger, G., Edwards, J., Ouziel, I., Huntley, B., Shwartsman, S., Durham, D. M., Anderson, A., and LeMay, M. (2014). Method and apparatus for fine grain memory protection. US20150378633.

Serebrin, B. and Haertel, M. (2008). Alternate address space to permit virtual machine monitor access to guest virtual address space. US20090187726.