Using The Virtualization Exception to Achieve High-Performance Virtual Machine Introspection

***Abstract*—Virtual machine introspection (VMI) is a promising technology capable of addressing security threats in a virtual machine (VM), by acting from the hypervisor (HV), below the virtual machine (VM)'s operating system (i.e. guest OS). VMI mostly relies on memory virtualization mechanisms, using second-layer translation tables (SLAT) to enforce access restrictions on certain areas of the VM's physical memory, including the page tables managed by the guest OS. Consequently, while executing the VM, the CPU will generate exceptions on any in-VM memory access not complying with the restrictions imposed by the VMI, switching from the VM to the HV (in a so-called "VM-exit"), giving the VMI module the possibility to analyze the faulty memory access and take the needed decision to protect the VM. While very simple, such a protection strategy could suffer significant performance penalties, as a large number of VM exits due to memory protection violations could be generated inside the page tables, most of which are irrelevant to virtual machine introspection (VMI), for example, accesses made by the hardware page-table walker when setting the accessed or dirty bits. We propose an approach to consistently reduce the number of irrelevant memory-related VM exits: by using the virtualization exception (#VE) extension of recent Intel processors, the in-guest faulty memory accesses can be handled directly inside the VM (in a HV-controlled kernel driver), filtering them out (i.e. discarding) if irrelevant for the VMI, while calling the HV (i.e. generate a VM-exit) only for the remaining ones. The in-guest filtering agent is protected against attacks from a compromised VM, by isolating it inside a separate guest physical address space, different by and inaccessible from the one used by the VM while running its own code. We implemented our #VE-based solution in the free and open source Xen hypervisor, obtaining performance improvements between 30% and 80% for the applications protected by our VMI module.**

***Index Terms*—hypervisor (HV), virtual machine introspection (VMI), virtualization exception (#VE), EPT violation, VM exit, vmfunc, performance improvements**

1. Introduction

VMI has been proposed by Garfinkel and Rosennblum in 2003 [1] as a security solution that is completely isolated from the monitored system. The main advantage of VMI is that it lies in the host, virtualization operating system (OS), the so called *hypervisor (HV)*, outside the guest OS running inside the protected *virtual machine (VM)*, and thus it is isolated from certain types of attacks, while also having an unaltered an complete view of the VM's memory.

Since then, a significant amount of research has been done towards extending the VMI technique and resolving its challenges, from which the most well known one remains the *semantic gap*, i.e. the complexity to correlate raw VM's memory with meaningful guest OS-specific data structures. Many different approaches were proposed, such as using invariants to identify the needed structures inside the VM's physical memory [2] [3] [4] [5] or more complex methods, such as graph-based approaches to identify the data structures and their relationship [6] or by using third party tools such as Volatility [7].

Security can be offered from VMI by using several approaches. Azab et al. [8] propose a method of ensuring VM integrity while also monitoring guest memory and critical events. SecVisor [9] is a tiny HV that prohibits unauthorized code from running inside the kernel of the guest VM. InkTag [10] ensures the integrity of user applications even on compromised systems. Approaches such as Ether [11] or MAVMM [12] were aimed towards providing means of dynamically analyzing malicious files in a virtualized environment. System call monitoring at the level of the HV [13], [14] was also proposed. Lycosid [15] uses hardware virtualization to identify hidden in-VM processes, Hooksafe [16] is aimed towards blocking kernel-mode rootkits, by monitoring the guest OS, while Dravkuf [17] performs real-time analysis of malicious samples. Recent VMI modules provide [18] real-time protection of live VMs for both kernel-space and user-space: such a scenario brings, however, many challenges and limitations [19], but the most obvious of these is the performance impact.

However the security target and the specific mechanisms implied, VMI mostly relies on memory virtualization mechanisms, using second-layer translation tables (SLAT) in HV to enforce access restrictions on certain areas of the VM's physical memory. Consequently, while executing the VM, the CPU will generate exceptions on any in-VM memory access not complying the restrictions imposed by the VMI, switching from the VM to the HV — in a so-called "*virtual machine exit (VM-exit)*", giving the VMI module the possibility to analyze the faulty memory access and take the needed decision to protect the VM.

When providing protection at process-level granularity, the VMI must also monitor any changes the guest OS perform on any protected process' page table (PT). In order

to do this, the VMI must write-protect specific processes' PTs. Because of this, the VMI protection strategy could suffer great performance penalties as a huge number of VM-exits triggered by memory protection violations could be generated, though most of them irrelevant to the VMI protection policy, like for example the settings of the access bit of all the in-guest PT entries inspected by the CPU page-walker during memory translations.

In order to improve the performance of a VMI module, the most notable research was proposed in [20], which makes a comparison between hardware assisted paging and shadow-paging, and proposes a method for dynamically switching between these two in order to achieve optimum performance. Other systems [21] aim to optimize network functions virtualization and thus improve network performance by leveraging new virtualization features present in the CPU.

We propose a novel strategy to reduce the high number of VM-exits generated by a protected VM: our experiments indicate that most in-VM memory accesses are made inside the guest PTs, and most of these accesses carry no useful information to VMI. Therefore, we aim to filter out these accesses directly inside the guest, avoiding this way costly and irrelevant VM-exits.

The main contribution of this paper is the creation of an in-guest filtering agent, which moves the processing of common and costly memory access violations from the HV inside the guest VM (in an HV-controlled and protected kernel driver), where they can be discarded with minimal performance overhead, if they are deemed irrelevant to the VMI logic, while calling the in-HV VMI (i.e. generate a VM exit) only for the relevant ones. We also propose the isolation of the in-guest filtering agent inside a dedicated guest physical address space, different by and inaccessible from the one used by the VM while running its own code. This way we mitigate attacks that may be carried out by an in-guest attacker.

Our VM-exit filtering algorithm makes use of the Intel *virtualization exception (#VE) extension*, which allows the CPU to report a memory access violation directly to the guest OS as a regular exception. This allows an in-guest filtering agent to get notifications about memory permission violations without generating a VM-exit. For the in-guest filtering agent protection, we use the Intel *VM functions technology (VMFUNC)*, which enables it to switch into and out of the isolated physical address space view created for its protection, without generating a VM-exit. The main contributions of this paper are:

- analysis of VM-exits caused by memory protection violations and identification of the most common ones and their main performance issues;
- implementation of an in-guest filtering agent, based on #VE, for improving the performance of memory access violation handling;
- a protection strategy of the in-guest filtering agent, using multiple physical address space views and the VMFUNC technology to switch between that views;

To the best of our knowledge, this is the first time the #VE and VMFUNC extensions are used to achieve high performance memory introspection, by filtering out these accesses directly inside the guest.

The paper is structured as follows: Section 2 offers an overview of the virtualization and VMI mechanisms, Section 3 illustrates the current performance limitations when dealing with VMI, Section 4 describes our proposed solution aimed to significantly improve the performance of VMI, while Section 5 shows our results. Finally, Section 6 shows some future research directions, while Section 8 draws the conclusions.

2. Virtualization and VMI Overview

2.1. Virtualization Basics

Hardware virtualization support, like the Intel Virtualization Extensions (VT-x) technology we refer to in this paper, is a CPU extension [22, Vol. 3, ch. 23] that allows efficient and secure virtualization of hardware resources, while honoring the Popek and Goldberg virtualization requirements [23]: *fidelity*, *safety* and *performance*. At the core of the VT-x technology lies the *Virtual Machine Control Structure (VMCS)*, which holds all the necessary information for both the virtualization hardware and the HV. A VMCS is assigned to each virtual CPU of each guest VM and contains all the control fields, host registers, guest registers and VM-exit information. A later addition to the VT-x technologies is the *second-layer translation tables (SLAT)* mechanism, or *Extended Page Tables (EPT)* on Intel, as we will refer to SLAT from here on. This technology allows the virtualization of the physical memory by introducing a second level of translation beyond the regular, legacy one: virtual addresss (VAs) are now called *guest virtual addresss (GVAs)*, and they translate via the guest PTs to *guest physical addresss (GPAs)*, which in their turn are translated via the EPT into *host physical addresss (HPAs)*.

The EPT is similar in structure to the PTs, and Figure 1 illustrates the translation of a GVA to a HPA. The EPT allows the HV to control guest physical to host physical memory mappings and access rights from outside the guest OS and independently of it. The access rights for a given host physical memory page are deduced from both the guest PTs and the EPT: in order to be able to write to a page, both the PTs and the EPT must have write permissions on that page. If the page is not writable inside the PTs, a *page fault exception (#PF)* will occur. If the page is not writable inside the EPT, an *EPT violation* will take place, which is a type of VM-exit, the mechanism by which the CPU notifies the HV that a virtualization event that may require HV's attention took place. The corresponding VM-exits will be processed by the HV and the VMI module, which will decide whether it is legitimate or not.

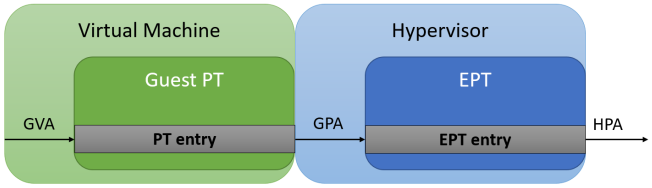


Figure 1: Address translation in a virtualized environment

2.2. #VE and VMFUNC

Recently, Intel added new capabilities to the virtualization hardware starting with the Broadwell architecture: the *virtualization exception (#VE)* [22, Vol. 3, ch. 25.5.6] and the *VM functions (VMFUNC)* extension [22, Vol. 3, ch. 25.5.5].

Once enabled, the #VE technology allows the conversion of certain EPT violations to a new type of exception. This is delivered inside the guest VM the same way a regular page fault or any other exception would be, via the *Interrupt Descriptor Table (IDT)*. The HV can control the physical pages for which an EPT violation or a #VE is generated by setting or clearing, respectively, the bit 63 inside the EPT entries: when this bit is 0, the entries are said to be *convertible*, because the CPU can deliver an #VE instead of an EPT violation. In order to use the #VE feature, the HV has to configure the *#VE information page*, which is a memory area accessible inside the guest, where the CPU will report the #VE specific information (similar to what is already reported during an EPT violation): the qualification and the faulty GVA and GPA.

The VMFUNC technology allows the HV to configure certain VM functions, which can be executed directly inside the guest VM, without generating a VM-exit. These VM functions are invoked by executing a new instruction, named VMFUNC, inside the guest. The value of the EAX register indicates which VM function ought to be executed by the CPU without generating a VM-exit. The only defined VM function so far is *VM function 0*, which is the *EPT-switching* function. It allows the guest VM to change the currently active EPT from a list of values preconfigured by the HV, a mechanism that allows the HV to control which EPT the guest could switch to, without generating a VM-exit for such a privileged operation.

2.3. VMI Basics

By having a *VMI module* that runs inside a dedicated VM, one can take advantage of the following properties:

1) *scalability*: a single VMI instance can protect multiple guest VMs;
2) *deployability*: because installing and configuring a single VMI VM ensures security for multiple other VMs;
3) *security*: the VMI instance running inside the HV or in a separate, dedicated VM, is inaccessible from other guest VMs.

The VMI module makes use of the CPU virtualization extensions and the HV in order to provide security for the running VMs. We focus on real-time, on-premise VMI, which must be capable of analyzing the behavior and the state of the running VMs with minimal performance overhead, while not sacrificing security properties. In order to provide security, the VMI module must first bridge the semantic gap, i.e. obtain relevant, meaningful information from otherwise raw VM's memory pages. Once relevant structures have been identified in the guest VM's memory space, the VMI module can employ protective measures to prevent malicious modifications of these structures: for example, it may prevent modifications to read-only memory areas. In our particular scenario, the VMI module is capable of providing protection for both the kernel components and the user processes present inside the guest VM.In order to do so, certain low-level events, such as kernel module load/unload, process creation/termination or memory allocation/deletion must be intercepted. The relevant memory areas can then be protected against specific attacks: a stack or a heap will be protected against code execution, while read-only code sections will be protected against modifications. All of these have one thing in common: the VMI module enforces memory access restrictions on critical areas of the kernel or process memory. User-mode protection implies monitoring user processes as soon as they are created and protecting them against code injections, malicious modifications inside user-mode libraries and malicious code executions outside legitimate modules.

In order to block such memory attacks, the VMI module makes use of the EPT: first, it has to identify each object of interest (processes, stacks, heaps, drivers, etc.), then it has to protect them by restricting, inside the EPT, their memory access. An EPT violation will then take place whenever an execution in the VM tries to access a memory page in a way that conflicts with its intended access rights established inside the EPT. The analysis of such events indicates the source and the destination of the fault and aids the VMI module into determining whether the access is legitimate or not. For example, when loading a kernel module, some legitimate write faults will take place when the kernel would fix the module's import table. Such legitimate events will be emulated by the VMI module. In contrast, if a write fault inside the code section of the module takes place after it was loaded, that is indicative of an attack and the VMI module can take the appropriate measures, e.g. block the write by skipping the faulting instruction.

In addition to protecting certain memory areas, the VMI module can also protect hardware structures, such as model-specific registers, control registers or the descriptor table registers.

2.4. Protecting the Memory

As we explained above, in order to protect the guest VM's memory against attacks, the VMI module uses the EPT to enforce restricted access rights on certain memory pages. However, the restrictions work only on the *guest physical pages (GPPs)*, which are translated using the EPT, and not for *guest virtual pages (GVPs)*, which are used

inside the guest VM. Since the OS and the applications inside the VM use only GVAs, the VMI module has to make sure that the correct GPA has been identified for any given GVA. This can be done by doing a page walk, i.e. parse the guest PTs, the same way the CPU would do when accessing the memory, in order to translate a GVA into its corresponding GPA. This step ensures that the correct GPA and GPP it is located in are identified for any given GVA, such that the identified GPP to be marked with the needed restriction on its access rights inside the EPT.

Most of the times, kernel memory will be non-paged, which means that the memory manager will never swap out those pages, and that the guest virtual to guest physical translation will always be valid and unchanged. However, certain areas of the kernel memory and the entire user memory space is pageable, which means that it may or may not be present in physical memory at any given moment, the memory manager being free to swap such pages in and out of the physical memory when needed. This creates a problem, because even if we properly identify the guest virtual to guest physical mapping initially, that mapping may change in time. In order to overcome this, we need to intercept the guest virtual to guest physical mapping modifications: this is done by marking the guest physical memory containing the guest PTs as non-writable inside the EPT, and thus intercepting any mapping modification the guest OS memory manager may do. Each such modification attempt would generate an EPT violation, which would be handled by our VMI module by decoding the old and the new PT values. This way, any change inside the guest PTs can be handled by the VMI module, before making it effective.

A *page table entry (PTE)* consists of several control bits, ignored bits and mapping bits. However, the VMI module is interested in only a subset of these bits, but modifying any of them would still trigger an EPT violation. In our particular scenario, the following bits inside a PTE (illustrated in Figure 7) are considered relevant: the present (P) bit (bit 0), the read/write (R/W) bit (bit 1), the user/supervisor (U/S) bit (bit 2), the page-size extension (PS) bit (bit 7), the execute-disable (XD) bit (63) and the mapping bits (bit 12-51). All the other bits (e.g. the access and dirty bits) are irrelevant from a security perspective, and intercepting their changing attempts is of no use to the VMI module. Even more, some of these bits are ignored by the CPU, and may be used by the guest OS's memory manager for internal tasks (such as page replacement policy or even as locks), also irrelevant from a security perspective.

3. Current Performance Limitations

The VMI module is event-driven, as it relies on events being generated inside the guest VM in order to decide whether a malicious attack is taking place or not. The types of events that can be intercepted range from hardware registers modifications (control registers, model specific registers or debug registers) to invalid memory accesses that trigger an EPT violation. Each such event generates a VM-exit, which will be handled by the HV and the VMI module, in order to decide whether it is legitimate or not. Such VM-exits carry a significant latency, caused by the:

1) hardware switch of CPU from VM to HV and back;
2) HV processing;
3) VMI processing;
4) event emulation.

As mentioned before, any restriction that we enforce inside the EPT would work for guest physical memory only. Since the guest VM's OS and applications work with guest virtual memory, we need to create a binding between these two. This binding will be the guest PTs that are used to translate GVAs to GPAs. Protecting a GVA implies protecting the GPA it translates to and protecting the guest PTs in order to intercept any change of the identified mapping (like swap operations). This ensures that we always have an updated GVP-to-GPP association for the GVP we want to protect and, consequently, can maintain the correct protection inside the EPT.

Protecting the guest PTs leads, however, to significant challenges. First of all, all levels of the multi-level structured guest PTs need to be monitored, as translation modifications can be done at any level. This means that for each code or data page that we want to protect, we need to intercept four PT pages as well (assuming 64-bit 4-level paging): PML4, PDP, PD and PT. Since the number of writes made by the guest OS inside the PTs is very high, especially on 64-bit systems where the virtual address spaces are large and more space (i.e. pages) is used for the PTs, the overall performance impact is increased, especially when the VMI module is protecting multiple processes, since more memory is monitored using the EPT. In order to assess the impact of monitoring the guest PTs, we first measured how many EPT violations were generated due to the OS modifying PTEs, in a regular system usage scenario, where two processes, i.e. Chrome and Firefox Internet browsers, were protected against attacks. Next, we counted how many of these PT modifications touched any relevant bits, as mentioned in subsection 2.4. The results are illustrated in Figure 2: more than 90% of the total number of EPT violations represent irrelevant modifications inside the PTs, made either by the OS memory manager or the CPU page-walker. Our main goal is to improve the VMI module performance by processing faster, or even avoiding, these common, yet undesired EPT violations. Examples of such irrelevant modifications include clearing the accessed or dirty bit or modifying portions of the guest PTE which are ignored by the CPU and are used by the OS for internal memory management tasks.

Secondly, we analyzed in more details the hardware page-walker, which we found to be an important source of performance impact, as each time a page-walk is be performed, the CPU sets the accessed and dirty bits, if required, in the relevant PTs. The CPU will set them whenever performing a page-walk, if they are not already set. However, the page-walker will set these bits atomically in all levels of the paging structures: for example, if the accessed

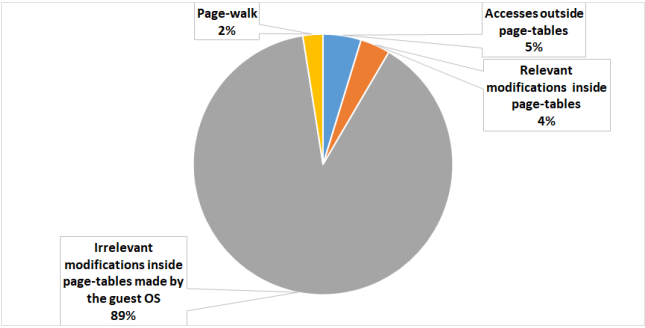


Figure 2: Distribution of EPT violations in a regular usage scenario

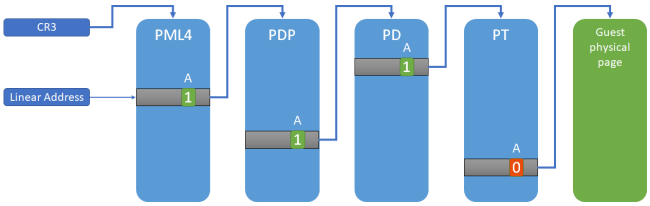


Figure 3: Guest PT hierarchy with the accessed bit set in all levels except for the PT

bit is set in all levels of translation except for the last level, the PT, the CPU would still try to set it atomically in PML4, PDP, PD and PT. Figure 3 illustrates this concept: given a virtual address, the accessed bit is set in all the higher level PTs (PML4, PDP, PD), but it is not set inside the last level PT. This means that when the CPU will perform the page-walk for that virtual address, it will try to set the accessed bit inside all levels, even if it is already set in all of them except for the last one. Since the VMI module needs to protect all levels of translation, this means that each page-walk that will set the accessed or dirty bit in any level will trigger an undesired EPT violation. In Figure 4 we can see that in high memory pressure scenarios, a very high fraction of the EPT violations will be caused by the CPU page-walker. The intensive memory usage scenario was induced by allocating so much memory so that the OS memory manager was forced to swap physical pages in and out of the physical memory. This was done by allocating more memory than the total amount of physical RAM installed on the testing system. Although the test was synthetic, it illustrates the memory manager's behavior pretty well in similar real-life scenarios.

4. Using #VE for High Performance VMI

We discussed so far the mechanics used by the VMI module in order to provide VM protection: at its core lies the EPT, which is used to enforce certain access restrictions on the VM's physical pages. In order to properly monitor those pages, the VMI module must also intercept writes inside the guest PTs, to ensure the proper GVA-to-GPA memory mapping is protected. Both the guest OS and the CPU trigger

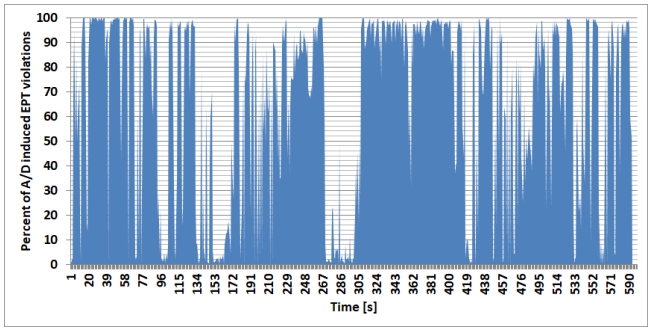


Figure 4: Histogram showing the fraction of the page-walker induced EPT violations, relative to the total number of EPT violations, in an intensive memory usage scenario

a lot of writes inside the PTs, which lead to a significant performance overhead.

In order to improve the performance of the VMI module, we propose using the #VE and VMFUNC technologies to filter the guest PT accesses by splitting the VMI functionality between two components:

1) a *filtering agent*, located inside the guest OS and
2) the *main VMI module* located outside the guest VM (e.g. in a dedicated security VM).

The separation of the VMI components is illustrated in Figure 5 and Figure 6. The in-guest filtering agent handles only the EPT violations generated by changes on the guest PTEs, filtering out the ones irrelevant for the VMI protection and sending the others through VM-exits to the main VMI module for further analysis. The later one handles EPT violations not filtered out by the in-guest filtering agent, EPT violations generated outside guest PTs and other types of VM-exits. In addition, the main VMI module also has the responsibility of configuring inside the EPT the guest PT's pages (1) *convertible*, so that the EPT violations triggered by changes on that pages to be delivered as #VEs inside the guest, or (2) *non-convertible*, once they need not be monitored anymore.

Beside splitting the VMI module in two components, we also propose creating *a new view of the guest VM's physical address space* (through a *new EPT configuration* or *view*, different than the one already associated to the VM) to place the filtering agent in, in order to protect it from the untrusted guest OS. The new physical address space view isolates the filtering agent from the rest of the OS, the same way processes are separated in different virtual address spaces via different PTs. In our case, the new guest VM's physical address space view is mapped on the same host physical memory already allocated to the VM, i.e. the same GPP is mapped in the new EPT view on the same HPA as in the original EPT, but with different access rights, such that the same GPA could have different access rights in the two different EPT views. The physical address space view created specifically for the filtering agent is called the *protected EPT view*, because only the filtering agent can be executed there, the entire guest OS being

marked as read-write, without execution rights, in order to prevent an attacker from running arbitrary code in the context of the filtering agent. The original VM's physical address space view, called the *untrusted EPT view*, is the one the guest VM normally executes in, i.e. the guest OS and applications has their normal rights, yet the filtering agent is made inaccessible there, in order to prevent its malicious modifications.

Since there are two distinct physical address spaces views, i.e. the untrusted EPT view in which the guest normally operates and the protected EPT view in which the filtering agent runs, we need to switch between them. This is accomplished by using the *VM function 0*, the EPT switch mechanism provided by Intel processors. This allows the in-guest filtering agent to execute the VMFUNC instruction in order to switch from the untrusted EPT to the trusted EPT and back. Because only the filtering agent and the VMFUNC trampoline are marked executable in the protected EPT, an attacker executing the VMFUNC instruction anywhere else in the guest OS or applications, while running in the untrusted view, in an attempt to switch to the protected view and run in it malicious code, will lead to an EPT violation (i.e. execution attempt from memory with no execution permission) being triggered on the next instruction. That EPT violation will be intercepted and handled by the VMI module by blocking the unauthorized execution attempt.

4.1. The In-Guest Filtering Agent

The in-guest filtering agent must be a kernel-mode driver, to be able to intercept and handle the #VE exceptions and filter out the irrelevant events for the VMI module. It can be pre-installed inside the guest VM as a regular kernel-mode driver or it can be installed dynamically by the main VMI module. In this paper, for simplifying the explanations, we consider it (including the #VE and trampoline pages, integral parts of the agent) to already be installed inside the guest OS. The filtering agent can be protected by the main VMI module, the same way other kernel drivers are protected against attacks, by preventing malicious modifications made to it. Typically this could be done by intercepting in the HV the guest OS's driver loading functions, identifying the memory allocated for a new driver and applying in the EPT the write permission restrictions on code and read-only data pages of that memory. Besides, as mentioned before, for our in-guest filtering agent's driver we completely restrict the execution permission rights in the untrusted EPT.

The VMI module has the task of setting up the entire environment for proper #VE and VMFUNC use in the filtering agent. This way, the attack surface is reduced, because all the initialization steps are done by the main VMI module when the filtering agent is loaded by the guest OS. If the filtering agent was not installed inside the guest, the VMI module would still work, because guest PT modifications would still generate EPT violations. However, the performance impact would be higher, since these EPT violations would not be filtered inside the guest. The logical steps performed by the main VMI module in order to enable the filtering agent are:

1) create the new, protected EPT view to provide a trusted environment the in-guest filtering agent could run in;
2) change in each CPU's IDT the entry corresponding to the #VE exception (i.e. at index 20) and make it point to the handler inside the VMFUNC trampoline page;
3) enable and configure the CPU's support for #VE and VMFUNC support, by programming specific VMCS fields;
4) configure the untrusted EPT's entries mapping the guest PTs to be convertible, so they generate a #VE instead of an EPT violation VM-exit.

4.2. The Alternate EPT View

The two EPT configuration and switching mechanism described above function as a memory isolation for the in-guest filtering agent, protecting it from the untrusted guest OS and applications while inside the untrusted EPT view, and providing it a trusted running environment while inside the protected EPT view.

In order to ensure that an attacker running malicious code (injected in a compromised guest OS or application) in the untrusted EPT view cannot tamper with the filtering agent or cannot run that malicious code inside the protected EPT view the following configurations must be prepared by the main VMI module:

- the filtering agent needs to be marked as non-accessible (i.e. no permission rights) inside the original, untrusted EPT view and read-write-execute inside the protected EPT view; this ensures that the untrusted guest OS and applications cannot tamper with the filtering logic;
- the other parts of the guest OS need to be marked read-write-execute inside the untrusted EPT view, but only read-write inside the protected EPT view; this ensures that an attacker would not be able to execute his code inside the protected EPT view, since only the filtering agent would be marked executable;
- in order to transition from the untrusted EPT view to the protected EPT view and back, a trampoline page is needed, that is marked as execute-only in both EPT views; this trampoline page contains the code responsible for setting up to proper transition by executing the VMFUNC instruction and thus switch from one EPT to another; being non-writable its contents cannot be tampered with and its only functionality, even accessible in both views, is its designed one.

An important aspect of the protected EPT view is that the entire guest OS and applications are marked as read-write inside of it, having no executable components other than the filtering agent and the VMFUNC trampoline page. This means that we cannot rely on OS functionality in order to carry our tasks, e.g. we cannot call memory management routines, because the kernel itself is untrusted and will be marked as being non-executable. Our filtering component has to be self-dependent and any low-level task, such as memory management, has to be implemented internally. Not having executable access to other components except for

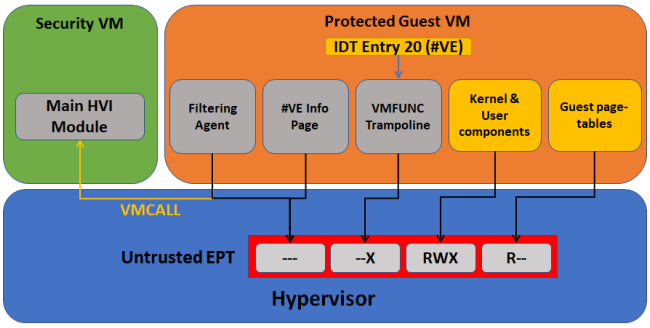


Figure 5: The mapping of various components inside the untrusted EPT view

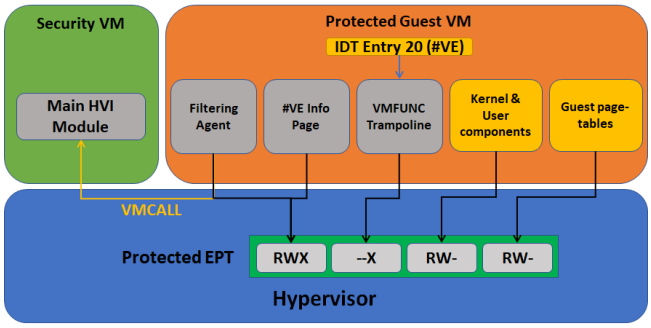


Figure 6: The mapping of various components inside the protected EPT view

the filtering agent is, however, beneficial from a security perspective, since an attacker cannot take advantage of executable mappings present inside the protected EPT view.

Once the protected EPT view has been created, the VMI module needs to configure the #VE and VMFUNC support, together with initializing the #VE information page described in subsection 2.2. In order to ensure that no one tampers with the #VE information page, this will be mapped as read-only inside the untrusted EPT view and read-write inside the protected EPT view. Though, while an EPT violation is converted to a #VE exception, the CPU can still write information describing that event in the #VE information page, because it accesses it directly using a HPA, which is not subject to restrictions in the untrusted EPT.

Figures 5 and 6 illustrate how various components are mapped inside the untrusted and protected EPT views. The kernel and user space components are mapped with full permission rights inside the untrusted EPT view and with read-write permissions inside the protected EPT. Actually, the full permission rights are subject to VMI policies, so in fact portions of the kernel or user space components may be restricted from certain accesses, like for example, stacks may not be executable and read-only code pages may not be writable. The filtering component together with the #VE info page will be mapped with no access rights inside the untrusted EPT view and with full rights inside the protected EPT. The trampoline page needs to be mapped execute-only in both EPT views, while the guest PTs will be mapped read-only inside the untrusted EPT view, but read-write inside the protected EPT view. The filtering agent communicate with the main VMI module through hypercalls issued by using the VMCALL instruction, a CPU instruction dedicated to the sole purpose of generating a VM-exit from a guest VM. This communication is needed in order to allow the filtering agent notifying the main VMI module about guest PT modifications which are worthy of attention.

4.3. Filtering Page Table Accesses

We use different criteria for classifying the in-guest PT modification accesses, which, due to the described EPT settings, will generate #VE exceptions handled by our in-guest filtering agent. If we consider the *source of the access*, we have (1) *software accesses* (those made explicitly by the guest OS' memory manager) and (2) *hardware accesses* (those made by the CPU page-walker). However, we also classify them with respect to how useful they are to the VMI protection logic, in which case we have (1) *relevant accesses* (such as the ones that modify control bits or the physical address inside the PTE) and *irrelevant accesses* (such as the ones made by the CPU page-walker or the memory manager, if they modify other bits then the ones monitored by the VMI module). More precisely, the in-guest filtering agent deals with the following combination of PT modification access types and takes the following decisions regarding where they should be handled:

1) hardware accesses, which are always considered to be of no use, i.e. irrelevant accesses, to the main VMI module and handled locally;
2) software accesses, which are further classified in irrelevant or relevant, based on the bits they change, and handled locally or transferred further, as VM-exits, to the main VMI module, respectively.

A CPU page-walker induced #VE exception is reported via a dedicated bit of the qualification field inside the #VE information page, so it is very easy for the filtering agent to identify that type of PT accesses. Handling them would normally need implementing a page-walk emulator to make sure the appropriate access and dirty bits are correctly set at all PT levels involved in the translation of the faulty GVA. However, since the filtering agent, running in the context of the protected EPT view, has read-write permission rights on all guest VM's memory, it can simply access the faulting VA in order to force a new page-walk to take place and let the CPU set the needed bits. We have to note that no #VE exception will take place due to such an operation, because unlike the untrusted EPT view that has restricted accesses on guest PTs, the protected EPT view in which our filtering component runs is unrestricted and allows writes to take place inside the PTs.

Handling all the other PT changes is done in a similar way, i.e. let the CPU make the required changes, but controlled by the filtering agent, and besides, announce the

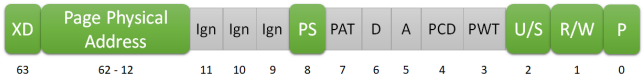


Figure 7: Relevant bits inside a PTE

main VMI module about the relevant events. First of all, the filtering agent will fetch the current value located inside the modified PTE, which we will refer to as the *old value*. Since it is running in the guest context, but within the protected EPT view, which allows writes to take place inside the PTs, the filtering agent can directly execute the instruction that triggered the #VE. As an effect, that instruction will do the intended modification inside the PTE. Executing the faulty instruction is done by copying its bytes inside the filtering agent, loading the original register values used by the guest OS when the #VE was triggered, and actually executing the instruction. Once the instruction finishes its execution, the filtering agent's registers are restored, and the *new value* located inside the modified PTE is fetched again. By comparing the old and new PTE's values, the filtering agent can decide whether the modifications need to be reported further to the main VMI module or not. Figure 7 illustrates which subset of the bits inside a PTE need to be monitored by the VMI module.

The complete PT access filtering algorithm is depicted in Algorithm 1. If the modification is due to the CPU page-walker, we can simply access the faulty GVA, which will end up setting the accessed or dirty bit. Otherwise, fetch the old PTE's value, execute the faulty instruction, and fetch the new PTE's value. If relevant bits are modified between old and new values, notify the main VMI module.

Algorithm 1 PT access filtering algorithm

```
1:  procedure FILTER(GVA)
2:      is_pw ← is_page_walker_access(GVA, ve_info)
3:      if is_pw then
4:          access(GVA)
5:      else
6:          old_PTE_value ← *GVA
7:          execute_instruction(guest_regs)
8:          new_PTE_value ← *GVA
9:          is_rel ← is_relevant_write(old, new)
10:         if is_rel then
11:             notify_vmi()
12: procedure EXECUTE_INSTRUCTION(guest_regs)
13:     instruction ← fetch_instruction(guest_regs.rip)
14:     saved_regs ← current_regs
15:     current_regs ← guest_regs
16:     call(instruction)
17:     guest_regs ← current_regs
18:     current_regs ← saved_regs
```

5. Tests and results

We conducted our tests on a host running Intel Core i7 7700K @ 3.6 GHz CPU, with 32 GB of RAM. The tests were ran in a VM running Windows 10 x64 RS6, with 2 virtual CPUs and 4 GB of RAM. The HV used to conduct the tests was XenServer 8.0, which is powered by the well known open-source Xen HV. In order to asses the performance improvement of the #VE filtering method, we used 2 types of measurement:

- **application startup time**, which measured how long it took for a specific application to start; we restrict our tests only on the application startup time and not on the entire application runtime, because the former is relatively independent of the different application's usage patterns; besides, during its startup, each application performs a lot of its memory's allocation and configuration operations, so it could be consider a phase of high memory accesses, thus relevant regarding the possible performance penalty induced by the VMI;
- **browsers performance test**, which measured different Internet browsers' performance in various activities, such as accessing an URL or opening a new tab.

Our VMI was configured to protect only a subset of the applications, which are commonly targeted by attackers. These include office applications (Word, Excel, Powerpoint), Internet browsers (Chrome, Opera, Firefox, Edge, Internet Explorer), email clients (Thunderbird, Outlook) and PDF viewers (Acrobat Reader). We restricted our VMI's protection only to the mentioned set of applications, because of the following reasons: (1) protecting all guest VM's processes would incur a high performance penalty, mainly because (2) there are many system processes that perform "illegal" memory operations for legitimate purposes, which would also trigger a lot of false positives, and finally, (3) the attack surface mainly consists in applications a user directly interacts with (as the ones we mentioned above) and the other system processes are attacked by lateral movement techniques, after compromising one of the applications in the attack surface.

The protection strategy employed for the mentioned applications included standard protective measures, such as: exploit protection (by preventing code execution outside legitimately loaded modules), injection protection (by preventing cross-process memory accesses) and hook protection (by preventing modifications to read-only code sections inside loaded modules).

5.1. Application Startup

First of all, we assessed the startup time of the protected applications by using the *Passmark AppTimer* benchmark. The methodology consisted of starting each application 5 times and computing an average startup time in 3 scenarios: (1) *baseline* (no VMI module present), (2) *VMI* (classic VMI case, i.e only the VMI module present) and (3) *#VE* (both the main VMI module and the #VE filtering agent are enabled).

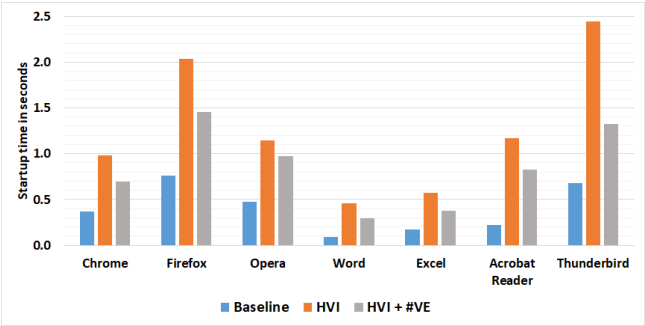


Figure 8: Application startup time with #VE optimizations

App	Baseline	VMI impact	#VE impact	Imp.
Chrome	0.36s	0.62s	0.33s	47%
Firefox	0.76s	1.27s	0.69s	46%
Opera	0.47s	0.67s	0.50s	26%
Word	0.09s	0.36s	0.20s	45%
Excel	0.17s	0.49s	0.20s	60%
Acrobat Reader	0.21s	0.96s	0.61s	37%
Thunderbird	0.67s	1.77s	0.65s	74%

TABLE 1: Application Startup Time in seconds and performance impact in percentage using #VE filtering

For The Internet browsers, which may contain multiple tabs, only a single empty tab was opened each time. For document viewers the applications was started without loading any document. In each scenario, except for the baseline, only the tested applications were protected against attacks. As one can see in Figure 8 and Table 1, the startup time improvement is significant for most of the tested application. Compared to the classic VMI, filtering PT accesses inside the guest improves application startup time by approximately 47%. This translates to an average impact of approximately 0.47s, added to the startup time using the filtering agent, compared to 0.87s, added when using only the main VMI module.

5.2. Browser Performance

We also ran a synthetic Internet browser test, in order to asses the performance and responsiveness of the Mozilla Firefox and Google Chrome browsers. The test consisted of measuring how long it took to open different addresses (i.e URLs), tabs and windows in several cases, such as: (1) opening an empty tab, (2) opening a tab with an URL, (3) opening a list of URLs in new tabs, (4) opening a list of URLs in a new windows and (5) opening a list of URLs in already existing tabs. The accessed URLs consisted of commonly accessed web-sites, such as *cnn.com*, *facebook.com* or *google.com*. The three tested scenarios are similar to the application startup test, i.e. baseline, VMI and #VE. Figure 9 illustrates the results on Google Chrome browser and Figure 10 illustrates the results on Mozilla Firefox browser. The performance improvement of the #VE filtering approach can be seen in details in Table 3 for Chrome and Table 2 for Firefox. The performance was improved by as much as 80% in some cases, with an average of 63%. The

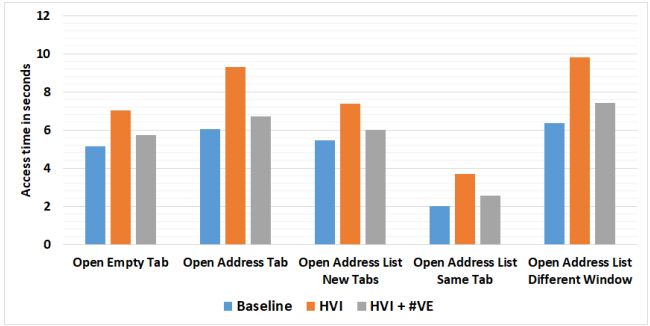


Figure 9: Google Chrome performance results with #VE optimization

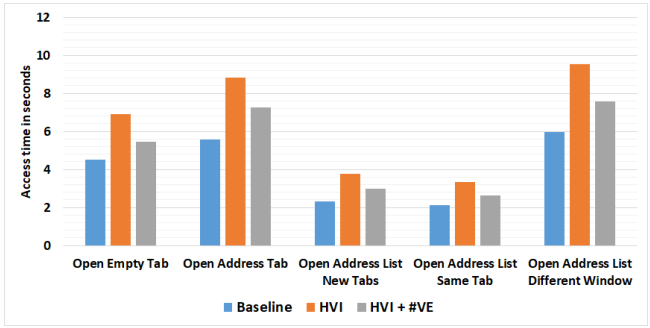


Figure 10: Mozilla Firefox performance results with #VE optimization

remaining average performance impact is 1.066s for Firefox and 0.686s for Chrome using the filtering agent, compared to 2.54s on Firefox and 2.44s on Chrome without the filtering agent.

5.3. Synthetic Paging Test

In order to asses to overall performance improvement on a per PTE modification, we created a synthetic testing user application protected by our VMI engine. That application triggered controlled changes of its monitored PT that were irrelevant to our VMI protection. The testing application al-

Test	Baseline	VMI impact	#VE impact	Imp.
Empty Tab	4.53s	2.38s	0.93s	61%
Address Tab	5.59s	3.24s	1.65s	49%
Address List New Tabs	2.33s	1.43s	0.64s	55%
Address List Same Tab	2.13s	1.18s	0.49s	59%
Address List Diff Window	5.97s	3.57s	1.62s	55%

TABLE 2: Browser performance impact in seconds and improvement in percentage for Firefox

Test	Baseline	VMI impact	#VE impact	Imp.
Empty Tab	5.13s	1.90s	0.61s	68%
Address Tab	6.04s	3.24s	0.66s	80%
Address List New Tabs	5.45s	1.94s	0.55s	72%
Address List Same Tab	2.01s	1.67s	0.53s	68%
Address List Diff Window	6.34s	3.47s	1.08s	68%

TABLE 3: Browser performance impact in seconds and improvement in percentage for Chrome

located some number of memory pages, with read, write and execute access rights, to force the VMI engine to monitor those pages against malicious executions and, therefore, to also monitor their corresponding PTEs, in order to determine their protection modifications or remapping operations.

After allocating memory pages, the application changed their caching attributes, which triggered changes on their PTEs, i.e. changing bits which are considered irrelevant by our VMI engine. With a traditional VMI strategy, each such operation would cause an EPT violation, when trying to change a monitored PT. However, our #VE-based filtering agent filters out all these writes, because they modify bits irrelevant for the application's protection. Our results, described in Table 4 and illustrated by Figure 11, confirm the significant performance gain. We varied both the number of pages (correspondingly, the number of PTEs) on which we triggered changes and the number of changes performed on each page. We performed tests on 1, 10, 100 and 1000 pages, corresponding to memory requests of $4KB$, $40KB$, $400KB$ and $4{,}000KB$, respectively, in order to determine the influence of the number of changed PTEs on our optimization mechanism. For each such a case, we also experimented with 100, 1,000, 10,000 and 100,000 operations changing each page's caching attributes. Thus, for instance, for the case of 1,000 pages, we measured our system's performance gains for 100,000, 1,000,000, 10,000,000, and 100,000,000, respectively, by comparing the time needed by our testing application to apply the changes to the allocated pages.

The case of acting on just one page was performed in order to identify the basic improvement our strategy brings when dealing with EPT violations due to irrelevant changes on a PTE over the traditional VM-exit-based strategy. The measured improvement was about 30%. We must note that, beside the PTE changes we triggered explicitly, the guest OS could performed some other changes on the same or other PTEs in our testing application's PT. Such additional PTE changes should normally not influence in any way our measured performance gains. We could note, however, that the greater the number of PTE changes, the greater the performance gains, especially for a greater number of allocated pages. We suspect that the longer the application's execution time (which is, obviously, implied by changing a greater number of pages and performing more such changes), the greater the chance for the guest OS to perform PTE changes that must be handled by the HV (like, for instance, page swapping, due to scheduling some other applications in the meantime). If all such changes were handled by the main VMI module, it would increase even more the execution time, due to the number of other VM-exits. On the contrary, when the irrelevant PTE changes are handled in guest VM by our #VE-based filtering agent, the execution time is reduced, and so is the number of other additional PTE changes (i.e. swapping operations) that should be handled by the main VMI module. So, having less VMI-handled PTE changes, when the #VE-based agent is on, compared to the case when it is off, lead to an even better performance than in the basic case, when mainly our induced PTE changes occurred.

6. Future Work

Each protected guest-virtual page has a PTE associated with it, which determines its translation, as can be seen in Figure 3. In 64 bit mode, a PT contains 512 such entries which control the mapping of a 2 MB range of memory. However, the VMI module may need to protect only a subset of guest-virtual pages in such a 2 MB range; because the EPT works with 4 KB pages, monitoring a single PTE means that we would still get a #VE for the modification of any entry in the given PT, even though they do not correspond to monitored pages. In the current implementation, all writes are treated the same, irrespective if they belong to monitored entries or not. Addressing this can be done by implementing a bitmap inside the filtering agent, which indicates, for any monitored guest PT, which entries belong to monitored pages and which do not. The bitmap would be updated by the VMI module, whenever adding or removing pages from protection, while the filtering agent would consult the bitmap whenever a #VE takes place. Modifications made inside entries which do not correspond to monitored pages would not be notified to the VMI module, thus further reducing the performance impact.

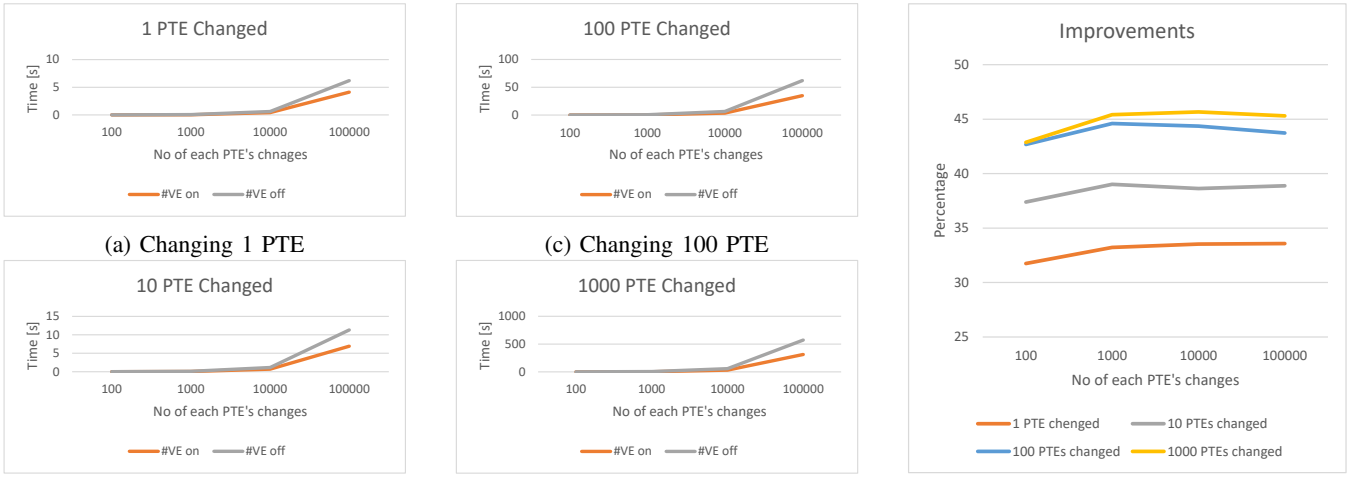
Currently, our solution works by filtering accesses made inside the guest PTs. However, the VMI module is capable of protecting other guest structures as well, some of which may be accessed very often, making their protection a very high performance burden. By using the approach of #VE filtering, new structures may be protected with minimal performance overhead: examples may include process and thread structures or security tokens. In addition, the filtering logic could be extended together with future #VE capabilities - for now, #VE supports only delivery of EPT violations. In the future, however, it may support delivery of new events, such as Control Register or Model Specific Register accesses, making the filtering more extensive and further increasing the VMI performance. While currently the EPT events are the most common, in the future, new protection mechanisms could require new events to be delivered as #VE in order to achieve high performance.

Our solution was implemented for 64 bit Windows OSs only, mainly because these are the OSs our VMI module mostly focuses on, and because 64 bit Windows make heavy use of the PTEs even for other purposes, such as locks; however, this method can be used on Linux or 32 bit Windows OSs as well, since there is no technical impediment - with some engineering, the #VE filtering agent can be adapted to run on both Linux or 32 bit OSs. In addition, our tests targeted only long-mode 4-level paging (since 2-level and 3-level paging are present only on 32-bit OSs), and we expect the performance improvement to be even higher with 5-level paging, since the number of monitored PTs will increase.

Finally, the #VE filtering approach can be combined with other technologies, such as the Sub-Page Permissions (SPP) [22, Vol. 3C, ch. 28.2.4], which reduces the size of a monitored area from the regular 4 KB pages to 128 bytes sub-pages. SPP works for write accesses only, and could be used to augment the PT monitoring or the protection of other

No of PTE changes	No of changed PTEs											
	1			10			100			1,000		
	Exec. time [s]		Imp.	Exec. time [s]		Imp.	Exec. time [s]		Imp.	Exec. time [s]		Imp.
	#VE on	#VE off	[%]	#VE on	#VE off	[%]	#VE on	#VE off	[%]	#VE on	#VE off	[%]
100	0.0043	0.0063	31.75	0.0072	0.0115	37.39	0.0364	0.0635	42.68	0.3286	0.5752	42.87
1,000	0.0416	0.0623	33.23	0.0694	0.1138	39.02	0.3482	0.6285	44.60	3.1277	5.7293	45.41
10,000	0.4122	0.6202	33.54	0.6952	1.1329	38.64	3.4952	6.2827	44.37	31.2765	57.5583	45.66
100,000	4.1208	6.2039	33.58	6.9225	11.3259	38.88	34.8722	61.976	43.73	313.1023	572.3702	45.30

TABLE 4: Improvement of the #VE optimization over the running time of an application triggering changes on its PTEs



(a) Changing 1 PTE (c) Changing 100 PTE

(b) Changing 10 PTE (d) Changing 1000 PTE (e) Improvements

Figure 11: Improvements of the #VE optimization

read-only data structures, thus further reducing the number of #VEs.

7. Related Work

NEM [24] is the most similar work with ours: they also use #VE to avoid VM-exists due to EPT violations and VMFUNC mechanism to switch between multiple protection domains' EPTs. Though, there are some notable differences between NEM and our solution. Firstly, they do not identify performance bottlenecks due to EPT violations, neither really evaluate any improvements gained by using the #VE mechanism. Secondly, regarding the multiple EPT based protection, it is not clear if an attacker cannot trick their system by executing attacker's code in their "View 2" EPT. Finally, NEM's scope is process debugging based on a HV controlled instruction-by-instruction execution strategy.

There are few other solutions EPTI [25], Hodor [26], CrossOver [27], SeCage [28], Xen's altp2m [29] that use multiple EPTs and the VMFUNC-based EPT switching for providing isolation between different protection domains without the HV intervention in a very similar way we do, but generally for other purposes.

EPTI [25] uses multiple EPTs to protect guest OS's memory from Meltdown attacks. They also need to write-protect parts of the in-guest PTs (i.e. the PML4 level) in order to detect OS's changes on them and try to avoid trapping in the HV on CPU's PT walks due to A/D bits changes. Though, their optimization technique is different by our: they map the monitored user-space PTs' GPA as writable in EPT (to allow the CPU walking mechanism changing the A/D bits without triggering EPT violations), but change the guest OS's GVAs of those PTs to GPAs write-protected in EPT. Their solution is not so general like ours: they must walk the kernel-space PTs to find out the needed GVAs, which implies write-protecting the kernel-space PTs in order to detect possible changes of the identified GVA-GPA mappings. So, their strategy is not applicable to the OS's own PTs. Besides, they do not explicitly measure the effect of their A/D bits changing optimization, but only in combination with other optimization technique making sense for their purpose. Even in such a case, the best performance improvement they gained was about 4%, while we obtained, on average, an improvement of about 40%-50%, and even 80% in best cases. Their protection is indeed limited (just the guest OS's PTs) compared to ours (more user-space processes' PTs), which proves that our optimization is really useful in more general cases. Our #VE-based mechanism is even more general, avoiding useless VM-exits on other events not just the A/D bits changes.

Hodor's [26] scope is to isolate multiple protection domains using more EPTs and VMFUNC for EPTs switching. They use a strategy similar to ours to avoid an attacker's attempt to call VMFUNC in unauthorized places. Though, the protection of our in-guest #VE-based filtering module is just part of our proposed solution.

CrossOver [27] proposes an efficient "world switch" (e.g. user to kernel space, VM to HV, VM to VM) mechanism decoupling the authentication (done by hardware) from the authorization (done in software by the callee). They simulated their CrossOver mechanisms by using the VMFUNC virtualization support to switch between different EPTs (i.e. different worlds) without HV intervention. Our in-guest #VE-based filtering module's protection is just a particular, simplified case of their CrossOver method, though we want to create isolated environments in the same VM.

SeCage [28] uses a similar multiple-EPT scheme like us to protect the secrets of user applications running in a VM in cases of guest OS and application itself being compromised. They suppose a trusted HV, like we do, and also use the Intel VMFUNC's EPT switching mechanism in order to avoid the overhead of multiple VM-exists. While their in-guest code and data protection strategy is similar to ours, our main goal is different: while we focus on performance improvements, they strictly focus on protection strategies.

ELI [30] proposes a strategy to avoid VM-exists due to hardware interrupts generated by I/O devices directly assigned to a VM. They configure the VMCS not to generate VM-exits on interrupts and replace the guest OS's IDT with a shadow one, which allows the guest OS directly receive and handle the needed interrupts, though forcing the others to trigger VM exits. Even if ELI uses an in-guest virtualization event handling mechanisms, like we do, their scope is different, i.e. reducing the overhead of VM-exists generated by hardware interrupts, while our is to reduce the overhead due to EPT violations. They do not use any in-guest handling code, even if they mention the possibility to do so, by shadowing the in-guest interrupt handling routines instead of the IDT. Though, they do not describe any protection mechanism implied by such a strategy.

The strategy in [31] is similar to ours only in the sense they try reducing VM-exists, though the optimization mechanisms are implemented in HV, having no in-guest handling agent.

8. Conclusions

We presented a new approach towards improving HV-based memory introspection performance, by using the latest virtualization capabilities introduced in Intel CPUs: #VE and VMFUNC.

The method relies on filtering guest PTs changes by using the #VE exception: since most of the EPT violations are generated due to PT changes and since most of them are of no interest to the main VMI module (e.g. page-walk accesses that set the accessed/dirty bits and certain memory-manager related modifications), they can be discarded inside the in-guest filtering agent, thus avoiding a costly EPT violation VM-exit. Only those accesses that modify the PTEs in a relevant way (e.g. adddress remapping or permissions changes) will be reported to the main VMI module.

We proposed and implemented a new algorithm in our main VMI module, by extending it with an in-guest filtering agent, which runs inside the guest OS and handles the #VE exceptions. In addition to providing filtering capabilities, the #VE filtering agent is isolated from the rest of the OS in a separate physical address space view. Thanks to the VM function technology, the #VE filtering agent can switch in and out of this protected physical address space view by using the VMFUNC instruction, without generating a VM-exit.

In order to asses the performance of our solution, we conducted tests using the Xen HV, in three different scenarios. We were able to reduce the startup time of protected processes by almost 50%. The Internet browsing performance was improved by 55% to 70%. The total number of eliminated EPT violations was measured to be close to 90%.

We concluded that the #VE and VMFUNC technologies are extremely promising and also proposed other research directions which may further improve the performance of VMI strategy.

References

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

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

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

[4] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

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

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

[7] J. Hizver and T. Chiueh. Real-time deep virtual machine introspection and its applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 3–14, New York, NY, USA, 2014. ACM.

[8] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *ACSAC '09 Proceedings of the 2009 Annual Computer Security Applications Conference*, pages 461–470, 2009.

[9] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, 2007.

[10] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: secure applications on an untrusted operating system. *SIGPLAN Not.*, 48(4):265–278, March 2013.

[11] 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*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[12] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, volume 0, pages 441–450, Los Alamitos, CA, USA, December 2009. IEEE.

[13] J. Shi, Y. Yang, and C. Tang. Hardware assisted hypervisor introspection, 2016.

[14] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-Based System Call Tracing for Virtual Machines. In *Proceedings of the 6th International Conference on Advances in Information and Computer Security*, IWSEC'11, pages 96–112, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] S. T. Jones, A. C. Arpaci Dusseau, and R. H. Arpaci Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 91–100, New York, NY, USA, 2008. ACM.

[16] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[17] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 386–395, New York, NY, USA, 2014. ACM.

[18] Andrei Luțaș, Adrian Coleșa, Sándor Lukács, and Dan Luțaș. U-HIPE: Hypervisor-Based Protection of User-Mode Processes in Windows. *Journal of Computer Virology and Hacking Techniques*, 12(1):23–36, 2016.

[19] A. Lutas, D. Ticle, and O. Cret. Hypervisor Based Memory Introspection: Challenges, Problems and Limitations. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, ICISSP '17, pages 285–294, 2017.

[28] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619. ACM, 2015.

[20] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. Selective hardware/software memory virtualization. *SIGPLAN Not.*, 46(7):217–226, March 2011.

[21] Jun Nakajima. Xen as High-Performance NFV Platform. *Xen Project Developer Summit, Intel Corporation*, 2014.

[22] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, May 2019.

[23] G. C. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures, 1974.

[24] Jingjie Qin, Bin Shi, and Bo Li. NEM: A NEW In-VM Monitoring with High Efficiency and Strong Isolation. In *International Conference on Smart Computing and Communication*, pages 396–405. Springer, 2017.

[25] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. {EPTI}: Efficient Defence against Meltdown Attack for Unpatched VMs. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 255–266, 2018.

[26] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, 2019. USENIX Association.

[27] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. Reducing world switches in virtualized environment with flexible cross-world calls. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 375–387. ACM, 2015.

[29] Lengyel, T. Stealthy Monitoring With Xen altp2m. https://github.com/tklengyel/drakvuf/wiki/Xen-altp2m, 2016. Accessed: 2019-07-19.

[30] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[31] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 373–385, Berkeley, CA, USA, 2012. USENIX Association.