Proposed Processor Extensions for Significant Speedup of Hypervisor Memory Introspection

Andrei LUȚAȘ[1,2], Sándor LUKÁCS[1,2], Adrian COLEȘA[2], and Dan LUȚAȘ[1,2]

[1]Bitdefender, Cluj-Napoca
[2]Technical University of Cluj-Napoca
`{alutas,slukacs,dlutas}@bitdefender.com`
`adrian.colesa@cs.utcluj.ro`

Abstract. Hypervisor based memory introspection can greatly enhance the security and trustworthiness of endpoints. The memory introspection logic requires numerous memory address space translations. Those in turn, inevitably, impose a considerable performance penalty. We identified that a significant part of the overall overhead induced by introspection is generated by mappings of guest pages into the virtual memory space of the hypervisor. We show that even if we employ highly efficient software caching, the mapping overhead still remains significant. We propose several new x86 instructions, which can fully eliminate the mapping overhead from memory introspection techniques. We give performance estimates for and argue why we strongly believe the implementation of such instructions to be feasible. The introspection logic also relies on monitoring guest page tables. Here we identified a second important performance overhead source, showing that numerous VM-exits induced by EPT violations are caused by the CPU updating page table A/D bits. We propose a set of simple x86 architectural modifications, that can fully eliminate this overhead.

Keywords: hypervisor, memory introspection, memory mappings, new x86 instructions, access/dirty bits

1 Introduction

Memory Introspection (MI) can be roughly defined as the process of analyzing a guest VM's memory from the hypervisor level. MI can be used to enforce the integrity of in-guest components or to detect a wide range of attacks against the OS kernel or user mode applications. MI scores a growing number of research in academia [18, 9, 10, 14, 23, 22] and widening adoption by industry [28, 11].

While having numerous advantages over conventional in-guest security solutions, like isolation and transparency, MI needs to solve two key challenges. The first one is the *semantic gap* [6, 10], briefly described in Section 2.2.

The second challenge arises from the fact that running MI logic outside the guest implicitly imposes running it inside a *separate virtual memory address space*, without direct access to the virtual memory space of the guest OS kernel or user mode processes the MI is protecting. For each guest memory area that needs to be accessed and analyzed by the MI logic (e.g. an instruction, a simple DWORD operand, a guest OS kernel structure etc.) it needs to parse, interpret and create paging structures that allows it to access the in-guest data from the hypervisor's virtual memory address space. While creating translations and mappings across various memory address spaces is a frequent operation in widely used operating systems, such as creating shared memory between two processes or mapping memory pages from the kernel into user space, the problem and impact of address space translations is exacerbated in memory introspection scenarios. During a typical MI analysis process we need to frequently access numerous and usually small sized in-guest data structures, in most cases only for a very short period of time, like for instance, only for the time it takes to execute a couple of "*if*" instructions. We show that such a usage pattern imposes a considerable performance penalty on the overall memory introspection process.

Besides the two key challenges, there are limitations imposed by the architecture and implementation of the underlying platform's support for hardware accelerated virtualization (in our case Intel x86) and/or by the guest VM's OS. One such limitation appears when the MI logic decides to perform detailed monitoring of a guest VM's paging structures. Such monitoring can be achieved by write-protecting the guest page tables at EPT level, but doing so, numerous unwanted VM-exits will be generated when the processor regularly updates the accessed/dirty (A/D) bits of the guest page tables.

The main contributions of this paper are:

– we identify guest-to-hypervisor memory mappings as being responsible for a significant part of performance overhead induced by MI (Section 3.2);
– we show that even with very good software caching (Section 3.1) there is still significant room to improve performance of guest-to-hypervisor memory mappings, thus there is a valid reason to consider hardware based speedups;
– we show that A/D-bit update is a key source of overhead, limiting efficient MI on Intel x86 platforms (Section 3.3);
– we propose several new x86 instructions (Section 4.1) that can fully eliminate the memory mapping overhead from memory introspection, propose small architectural changes that can fully eliminate the A/D-bit update overhead (Section 4.2), argue why it should be feasible to implement them on future x86 platforms (Section 4.3) and present performance estimations based on synthetic tests (Section 4.4).

During our research we did not explicitly seek to propose new processor instructions. We were focused on the speed-up of the MI logic and the underlying hypervisor. The feasibility of new instructions came when we realized that during the effective analysis process the MI logic from the hypervisor repeatedly needs to do very complex steps in order to access guest memory, while for the processor (just a couple of instructions before, while running in the guest) accessing the very same memory was a simple and straightforward thing to do. We initially noticed the overhead induced by A/D-bit updates by basic profiling, and subsequent detailed analysis revealed its significant impact.

1.1 Our use-case scenario

We performed our work on a production-grade proprietary thin-layer security hypervisor, built upon Intel x86 hardware virtualization acceleration technology. Our intended final use-case scenario imposed the MI logic to be capable of synchronously securing live Windows client endpoints (e.g. such as a typical office PC or home laptop running Windows), providing extensive protection for both the OS kernel and user-mode applications. We need to underline that such a use-case scenario requires both a very low overhead solution (the analysis is done on-the-fly for the running VMs) and not relying on in-guest modules at all (due to security reasons we assume the guest to be potentially malicious). Thus, our scenario excluded from the very beginning several traditional MI related approaches known in the prior art (such as, but not limited to, binary translation, shadow paging, asynchronous and/or snapshot based processing, source code level enlightenment of guest OS or that of the monitored applications, or relying on hardware memory acquisition).

While it is beyond the scope of this article to describe in details what our MI method covers, in order to place in context the measurements, we indicate some of the key capabilities. In kernel mode we mainly protect kernel and driver code sections and function tables (such as the IATs, SSDT, IRP dispatch tables and so on). In user mode we protect code sections, stacks, heaps, among others. Our method heavily relies on detailed monitoring of gust page tables with EPT interceptions on all 4 hierarchical levels. We use generic detection logic to identify events like code injections, function detouring or malicious code unpacking. We apply protection on the most critical user-mode applications, such as Adobe Acrobat, Microsoft Office or web-browsers, as those are key targets for the most prevalent malware and cyberattacks today.

2 Memory Introspection on x86 Platforms

2.1 Hardware Accelerated x86 Virtualization and Security

Although hardware accelerated x86 virtualization [16] was not specifically designed for security, because it provides strong isolation, it can be used to efficiently isolate a security solution from a possibly malicious environment or to isolate several execution environments from each other [20, 21]. Virtualization can be used in numerous ways to enhance security, such as providing secure execution environments [31, 8], do malware analysis [9, 30], provide integrity protection or attestation, among others. Lately, beside academic and open source community research, also traditional security solution vendors [28, 29, 11, 13] and security startups embraced virtualization based security [3].

Intel implements hardware accelerated second level address translation (SLAT) technology in their processors, termed Extended Page Table (Intel EPT), since 2008. Using it a hypervisor can efficiently control the physical memory seen by a virtual machine and specify read/write/execute permissions with 4KB page level granularity. The introduction of the EPT was essential to support efficient implementation of MI based security solutions.

2.2 Memory Introspection

Using Virtual Machine Introspection (VMI) [14] or hypervisor-based Memory Introspection (MI) we can analyze from the outside the contents of a guest VM, either suspended or in execution, observing not only objects inside the guest's memory space (like processes, modules, heaps, stacks, threads), but also events in real-time (such as the creation of a new process, allocation of memory or alteration of paging structures managed by the guest OS). While MI can have many other useful applications, the most widely researched area is to use MI to enhance security [18, 9, 10, 19, 26, 23], which led to its adoption by industry [28, 11, 27]. The two key security supporting strengths of MI are (*1*) *isolation*, by which the security module can be isolated by hardware from the possibly malicious or under-attack guest (e.g. an attacker inside the guest does not have access to the MI module running outside the guest, see Figure 1), and (*2*) *transparency*, which ensures that a MI solution can analyze the content of the guest without the need to alter it (e.g. without the need to use and rely on in-guest filter drivers, hooks or similar).

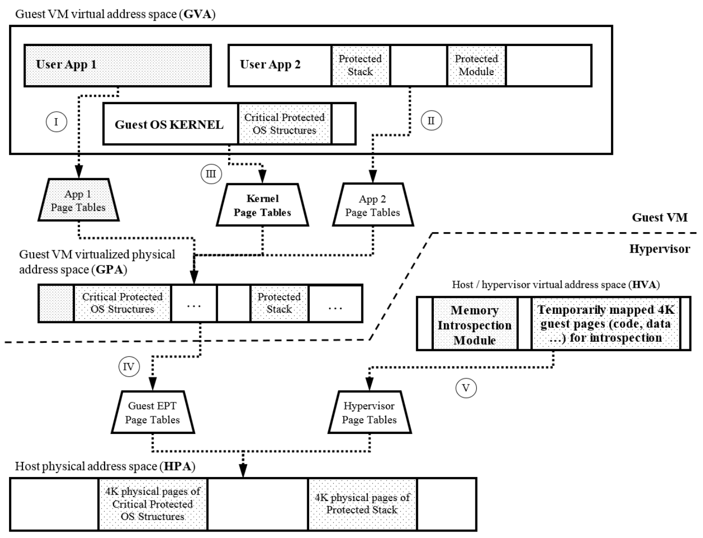

Guest VM virtual address space (GVA)
User App 1
User App 2
Protected Stack
Protected Module
I
Guest OS KERNEL
Critical Protected OS Structures
II
III
App 1 Page Tables
Kernel Page Tables
App 2 Page Tables
Guest VM
Hypervisor
Guest VM virtualized physical address space (GPA)
Critical Protected OS Structures
...
Protected Stack
...
Host / hypervisor virtual address space (HVA)
Memory Introspection Module
Temporarily mapped 4K guest pages (code, data ...) for introspection
IV
V
Guest EPT Page Tables
Hypervisor Page Tables
Host physical address space (HPA)
4K physical pages of Critical Protected OS Structures
4K physical pages of Protected Stack

Fig. 1: Memory Introspection. General setup and memory address translations involved

A key challenge that MI needs to solve is the so called semantic gap [6] between what the in-guest OS sees and what the MI running outside the guest, in the hypervisor, can see. In a rough approximation, we could say that an in-guest OS kernel and security solution, using well known OS provided APIs, can see and interpret the guest VM's memory as processes, threads, modules, heaps, stacks and various other, semantically rich and interconnected structures. In contrast with this, an out-of-guest, hypervisor level MI might see on the same VM only just a huge sequence (several gigabytes) of physical memory bytes, split into a 4KB pages, plus some processor registers to start the analysis with.

In order to bridge the semantic gap, a MI solution needs to analyze the contents of the guests' memory and understand its contents. This way, data structures, code sections and other objects can be identified inside the guest OS. Figure 1 illustrates the place of the MI module, as well as all the guest and hypervisor elements involved in the introspection process. Eventually, using such analysis of the raw physical memory, the MI can reconstruct the entire image of the OS by determining where the kernel is loaded, where the drivers, heaps, stacks, processes and threads are. In addition, information about user-mode objects can be inferred and protection can be provided for user-mode code and data as well. Once the semantic gap has been bridged, the MI can provide protection for many objects that lie inside the guest memory. First of all, inside the kernel-space, objects such as the kernels and drivers code sections or function tables can be write protected using EPT page permissions (set in Figure 1, item IV). To achieve this, the MI logic needs to determine their detailed GVA-to-GPA mappings (item III). Similarly, objects such as the stacks or heaps can be protected against execution using EPT, in both user and kernel memory space. To achieve this, the MI logic needs to analyze numerous GVA-to-GPA mappings (such as item I and II). Every time a violation of EPT access rights takes place (for example, the guest tries to write inside a page that is flagged as non-writable), the processor generates an EPT violation, which causes a VM-exit (a transition from the guest to the hypervisor). The hypervisor forwards the event to the MI module, which can analyze the attempt and decide whether it is legitimate or not. If it is legitimate, the instruction that triggered the violation can be either emulated or single-stepped. If the attempt is malicious, security measures can be taken. In its simplest form, the MI can decide to simply skip the instruction without doing any modification to the accessed page (e.g. this way could avoid setting a kernel hook). In another example, the MI logic can identify the process trying to do the illegal action and can terminate it.

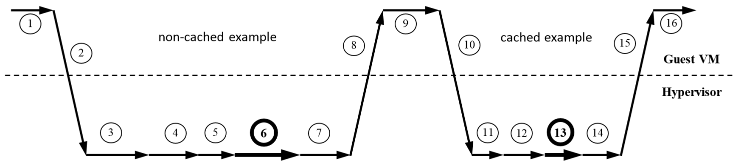


Fig. 2: Useful work steps (6 & 13) in the exemplary execution of two successive memory introspection related EPT violation events

Figure 2 illustrates, in a very simplified way, two typical guest ⇔ host transitions performed due to EPT violations, as a response to MI configured EPT access rights (Figure 1, item IV). Initially, the guest is executing code during step *1*. An EPT violation takes place, which causes a transition from the guest to the hypervisor at step *2*. This may cause the MI to initially map the memory page containing the offending instruction at step *3* into MI accessible virtual memory (Figure 1, item V). The analysis of the instruction takes place at step *4*, and it is followed by the mapping of the page that has been offended during step *5*. The actual useful work performed by MI follows in step *6*, such as analyzing the write attempt and deciding whether to allow it or not based on its maliciousness. This is followed by the unmapping of the previously mapped pages during step *7*, after which a VM-entry can be performed during step *8*, resuming the execution of guest code.

As we will further describe in Section 3.1, we implemented software caches to speed-up many repetitive guest-to-hypervisor memory mappings. A simple example could be the execution of successive instructions from a page marked non-executable in the EPT tables. In such a case, it is useful to avoid remapping the same guest page for each successive instruction. In line with this, in another example, during guest code execution in step *9* another violation may take place that will cause a transition to hypervisor in step *10*. This time, mappings might be present in internal software caches, which can be used during step *11* to search for the offending instruction's RIP and similarly, to lookup already-mapped pages that contains the written page. The evaluation of the instruction takes place during step *12*, and it is followed by the useful MI logic during step *13*. It is important to point out, that using caches involves also maintaining them up-to-date, so, while mappings might take much less time compared to the previous example, we need an additional step *14* to handle software cache maintenance. Finally, a VM-entry can be done during step *15*, then the guest can resume its execution again in step *16*.

3 Problems and Limitations of Memory Introspection

3.1 Software Speedup of Guest-to-Hypervisor Memory Mappings

Employing software caching to speedup memory translation or instruction emulation is not new. Among others Zhang et al use [37] per-VCPU software caching to reduce the overhead associated with privileged instruction emulation.

In order to improve guest-memory accesses, our HV also employs several software caching layers. The first one is a decoded-instructions cache, which stores information about instructions located at certain addresses inside the guest. The second cache is a guest-mapping cache which stores translations information for guest-virtual and guest-physical pages. The third one is a GPA-to-HVA translation cache, which stores HV mappings for the most used guest-physical pages. Instead of the third cache we could have used alternative mechanisms, such as mapping the entire GPA space into the HVA space. Though, such an approach would incur additional memory consumption and be more difficult to maintain in a many-guest-VM environment. Based on our measurements, our caching mechanisms had a hit rate of more than 99%, offering a very significant performance improvement. Particularly, in the case of the last cache, the performance improvement obtained was over 400x.

3.2 Overhead of Guest-to-Hypervisor Memory Mappings

The total overhead of the steps the MI module performs when accessing the guest memory is illustrated in Figure 3. The measurements were made on a

DELL Optiplex990, 8GB RAM, i7-2600 CPU system, using both Windows 7 SP1 x64 and Windows 8.1 x64 and four different testing scenarios. Firstly, the impact has been measured in *idle* conditions, with all software caches enabled. Then, the measurement has been made using typical *light usage* scenario, which included Internet browsing, opening PDF and MS Office documents. During the whole process, not only the guest OS kernel, but the browsers, Acrobat Reader and office applications were also protected by the MI module. The same two scenarios have been repeated, but this time without any software caches. The execution time has been accounted using the *RDTSC* instruction. We measured the total ticks spent inside VMXROOT (ring -1) and the total time spent inside the operations illustrated in Figure 1 together with the total number of VM-exits. The average time spent inside the guest memory access functions was computed using the formula $\frac{100*t_f}{T}$, where t_f is the number of processor ticks spent in a measured function and T is the total number of processor ticks spent in VMXROOT. It is worth specifying that in our measurements a small part of the operations overlap. For example, translating guest-virtual memory also involves mapping guest-physical memory and mapping guest-virtual memory also involved translating guest-virtual memory and mapping guest-physical memory.

Windows 7 x64				
Action	Idle, caches	Usage, caches	Idle, no caches	Usage, no caches
	Percent of total time spent in host			
Map GPA	2.36	**12.42**	21.15	**27.26**
Map GVA	2.48	**6.12**	32.90	**27.32**
Unmap GPA	0.00	**0.13**	0.56	**4.92**
Unmap GVA	0.06	**0.08**	1.52	**1.39**
Translate GPA	2.15	**11.09**	3.55	**3.64**
Translate GVA	3.12	**9.57**	26.12	**22.65**

Windows 8.1 x64				
Action	Idle, caches	Usage, caches	Idle, no caches	Usage, no caches
	Percent of total time spent in host			
Map GPA	3.46	**10.94**	31.93	**27.87**
Map GVA	3.61	**5.69**	23.42	**31.51**
Unmap GPA	0.00	**0.13**	0.25	**3.98**
Unmap GVA	0.08	**0.07**	1.19	**1.69**
Translate GPA	2.58	**9.92**	3.76	**3.33**
Translate GVA	3.67	**9.20**	18.80	**26.06**

Fig. 3: Time consumed (percents of in-hypervisor time) by various phases during the handling of memory introspection related EPT violation events

In the light usage scenario the performance impact may be easily over 15% with caches and over 30% without. Considering the workload involved, we believe the actual real impact may be even much higher in practice. We explicitly didn't try to obtain much finer grained or detailed measurements. On one hand this would have not been possible without extensive changes to the hypervisor, which in turn would generate biased results. On the other hand, it can be clearly seen from our analysis that the magnitude of the results is really significant.

Figure 4 illustrates the performance impact for two representative user-mode process operations: heap creation and destruction. Firstly, a scenario without any caches involved has been tested. The last two scenarios involves all the caches (as described in Section 3.1) and a special, application-specific cache. This special cache was created to maximize efficiency by keeping the heaps-array page mapped. However, this cache could not deal with the location of the newly created heap, which was rarely the same, and it still needed to map at least one guest-virtual page. We can see that software caches offer a significant improve-

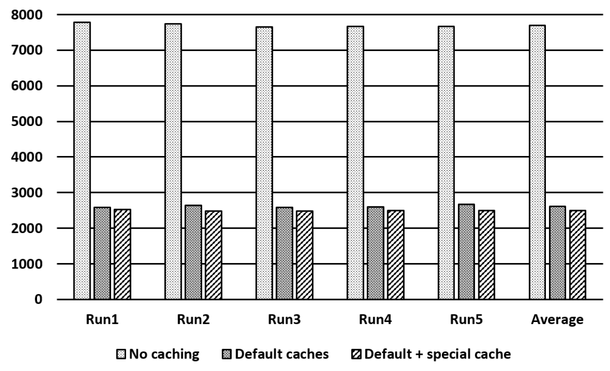


Fig. 4: Heap "create-destroy" software cache speedup of MI mappings (ms)

ment of about 3x. As the difference between the generic and the application-specific cache is very small, we conclude that it is very difficult to come up with any additional software optimizations, even if they are highly specific.

3.3 Overhead of A/D-bit Update Induced VM-exits

Looking at Figure 1, it is important to consider that GVA-to-GPA mappings are usually dynamic in time. The guest OS can arbitrarily alter them, such as by remapping a certain page to different GPAs due to the page swapping process. This imposes some important challenges for MI, as it needs to monitor not only the GPA addresses, but also the guest page tables that are used to perform GVA-to-GPA translations. This way, each time the guest OS performs a write to a page table, in order to update its content, the MI logic will be notified via an EPT violation, and thus, it can update its monitoring logic.

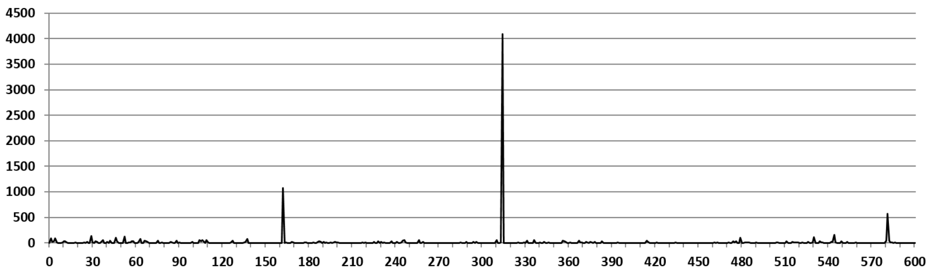


Fig. 5: Number of EPT violation VM-exits due to A/D bit updates (Idle)

This process can impose a considerable penalty if numerous writes take place on guest page tables. While at first, intuitively, it might seem that this is not the case, we need to factor in that on x86 platforms the CPU updates accessed/dirty bits regularly on guest page tables and any such update will generate an EPT violation that needs to be processed by the MI module, if and only if the MI logic is monitoring that particular page table. As the system load increases, so

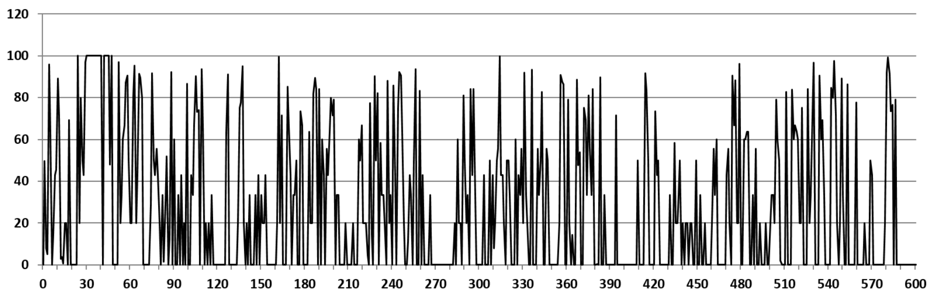


Fig. 6: Percentage of EPT violation VM-exits due to A/D bit updates (Idle)

does the number of paging operations (e.g. allocation, mapping, swapping), and also the overhead induced by MI. While not exclusively so, this phenomenon is highly specific to user-mode introspection, where the GVA-to-GPA mappings (steps I and II in Figure 1) are more frequently changed than in the case of kernel mode introspection (step III in Figure 1).

To analyze the impact of A/D bit updates done by the CPU page-walking on MI (we excluded the ones caused by the OS, which periodically clears those flags), we have carried out numerous tests on a Broadwell CPU based system, with 4 GB RAM and Windows 8.1 x64. Each test was run for roughly 600 seconds, and we have done sampling of EPT violation events at each second. Protected user-mode applications included Opera, Firefox, IE, Acrobat Reader and one custom memory-intensive application. We activated all MI protection mechanisms, both for user-mode and kernel-mode. The page table structures were not monitored entirely, only portions of them associated with protected guest memory areas.

We analyzed the impact of A/D bit updates in several different scenarios. The first scenario, as shown in Figure 5 and Figure 6 corresponds to an idle system that was running no other tasks than the OS itself. We consider this to be a baseline scenario. The easily observable spikes in this scenario are most likely related to periodic background OS processing.

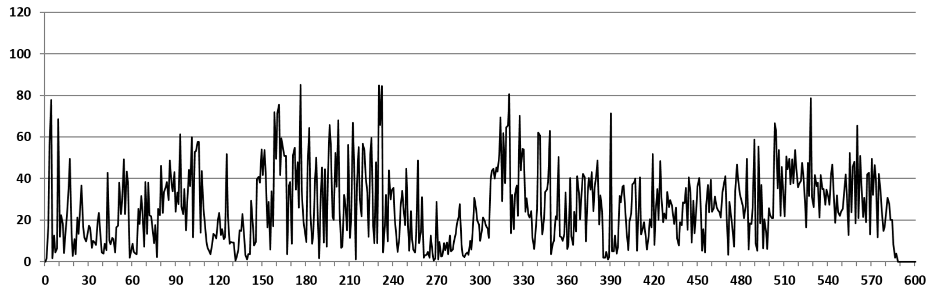


Fig. 7: Percentage of EPT violation VM-exits due to A/D bit updates (Light)

A second scenario is depicted in Figure 7. This corresponds to a typical light office workload, including browsing the internet with IE, playing YouTube videos, downloading files and opening documents with Adobe Reader. The spikes in this scenario are correlated with the creation of new MI-protected user-mode processes. This test clearly confirmed us that process creation regularly induces much bigger performance overhead for user-mode MI, as each new process requires the setup of complete memory spaces, thus heavily page-table (and implicitly A/D bit update) related workloads.

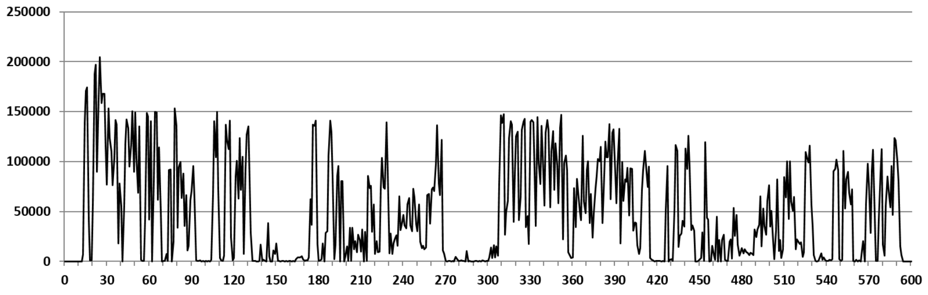


Fig. 8: Number of EPT violation VM-exits due to A/D bit updates (Intensive)

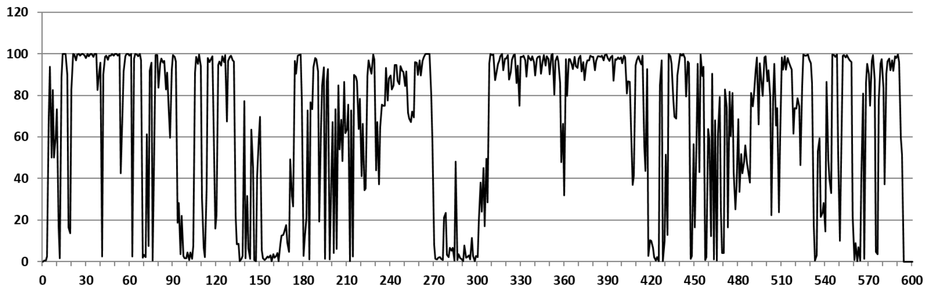
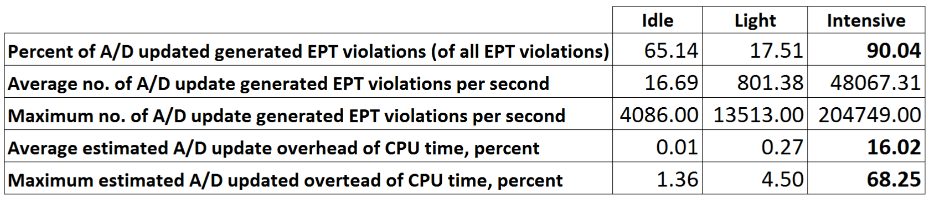


Fig. 9: Percentage of EPT violation VM-exits due to A/D bit updates (Intensive)

In a third case, as shown in Figure 8 and Figure 9, we analyzed the impact in the case of an intensive workload scenario. Here a test application was doing repeated allocations and processing of 2 GB memory chunks each 30 seconds, interleaved with starting a new IE, Opera or Firefox process every 10 seconds.

In figures 6, 7 and 9 we illustrate the distribution in time of the proportion A/D-bit induced EPT violations (as percents from all EPT violations). One can easily see that there are many cases when almost all of the EPT violations are caused by the CPU page-walker setting the accessed or dirty bits. In Figures 5 and 8 we illustrate the absolute numbers of A/D-bit induced EPT violations, showing that under intensive memory pressure, we may have up to 200,000 A/D-bit induced EPT violations per second.

Finally, Figure 10 aggregates all previous results, highlighting the high impact of the A/D-bit induced EPT violations and making it clear that they can be a major bottleneck. The most important slowdown appears on the creation of protected processes and under heavy memory pressure, accounting for up to 90% of all the EPT violations and up to over 60% of the the total CPU time.



	Idle	Light	Intensive
Percent of A/D updated generated EPT violations (of all EPT violations)	65.14	17.51	**90.04**
Average no. of A/D update generated EPT violations per second	16.69	801.38	48067.31
Maximum no. of A/D update generated EPT violations per second	4086.00	13513.00	204749.00
Average estimated A/D update overhead of CPU time, percent	0.01	0.27	**16.02**
Maximum estimated A/D updated overtead of CPU time, percent	1.36	4.50	**68.25**

Fig. 10: Estimated total CPU time impact of VM-exits due to A/D bit update

For the later one, we done our estimated normalizing the measurements to a 3 GHz clocked CPU and considering an average EPT violation handling time of 10000 clock ticks (without going into lengthy technical details, this includes the platforms round-trip, the HV context saving/restoring, handling software caches, decoding and emulation instructions, among others).

We underline that our results do not show that hypervisor based MI for user-mode applications would not be feasible at all, quite the contrary: for typical client endpoint doing everyday office work (as shown in the light workload scenario) the average overhead is acceptable. Our analysis confirmed our presumption that the biggest overhead is induced by workloads that put big pressure on the guest page table structures, indicating also the overhead's magnitude in unfavorable conditions. We plan to do more detailed analysis in the future, covering well standardized, server specific workloads also. We also point out for clarification that there is no relevant technical difference from the point of view of the MI logic between how user-mode and kernel-mode page tables are treated. What makes more costly for MI logic to monitor the page-tables of user-mode applications is the more dynamic nature of user-mode processes (e.g. start/stop, loading of modules) and much more dynamic virtual memory space allocation patterns, compared with kernel-mode code.

4 Proposed x86 Processor Extensions

4.1 New x86 Instructions for Direct Guest Memory Access

We propose the introduction of several new simple, yet very powerful x86 instructions, according to the following logic:

1. `READGPAB/W/D/Q <dest-reg>, [<src-gpa-addr>]`
2. `READGVAB/W/D/Q <dest-reg>, [<src-gva-addr>]`
3. `WRITEGPAB/W/D/Q [<dest-gpa-addr>], <src-reg>`
4. `WRITEGVAB/W/D/Q [<dest-gva-addr>], <src-reg>`

Those instructions differ in one essential way from a common MOV instruction: the translation of the operand's virtual address is to be performed in the virtual address space indicated by the *guest CR3 value* from the current VMCS (and not the current host CR3). The common step of GPA-to-HPA translation can be performed using the EPT pointer (EPTP) from the current VMCS, just as they are already performed by existing CPUs.

Handling of potential failures could be done easily. If the indicated GVA or GPA address is invalid or not present (or any other condition that would normally trigger a page-fault is encountered), the proposed instructions might simply set a common flag (e.g. carry flag) to indicate the error condition. This way, their usage should be very simple and straightforward.

4.2 Mechanism to Avoid VM-exits on A/D-bit Updates

EPT violations caused by the MMU page-walker setting the A/D-bit are generally not needed by the MI logic, although they induce a significant performance penalty. We propose simple extensions that would eliminate the performance impact caused by these exits.

Firstly, the A/D-bit induced EPT violations could be globally disabled via a control field inside the VMCS. This should inhibit any EPT violation that would be generated by the CPU page-walker when setting the A/D-bit. Such a mechanism would not restrict or alter the guest's functionality at all (e.g. the guest will be able to use page swapping as usual), and any effects would be visible only at the level of the CPU and of the EPT violation handling MI logic.

Secondly, the mechanism may be made more fine-grained by selectively flagging certain guest-physical pages as being "guest page tables". One bit inside the EPT page table entries could be reserved for this purpose: whenever the CPU page-walker sets the A/D-bit in a guest-physical page that has the "is guest page-table" bit set inside the EPT, an EPT violation would not be triggered (in other cases the exception shall be generated as usual). Using such a mechanism, the HV would have greater flexibility to control exiting on A/D-bit updates.

4.3 About the Feasibility of the Proposed Extensions

While we are not CPU designers, observing a set of relevant facts, we can still meaningfully argue on weather the proposed extensions are reasonable or not to be implemented. We assume as a general rule of thumb, that implementation complexity increases proportionally with functional complexity.

- We can be sure that in many cases translation information is already present inside the processor TLB for GVA or GPA addresses that cause EPT violations. This become clear if we consider two aspects. On one hand, since 2008 Intel already supports a feature called virtual processor IDs (VPIDs). When activated, the TLBs tag each cached memory address translation with a guest specific unique ID, with the ID 0 being reserved for the hypervisor. Later on, when the HV asks the processor to invalidate cached translations, it can specify individual VPIDs to specify the target of the invalidation. Thus, while executing code inside the HV, all translation cached in the TLB on behalf of the guest OS remain valid and present, unless the HV explicitly requires their invalidation. On another hand, when dealing with MI related EPT violations, we can safely suppose that most of the in-guest memory addresses involved are already in the TLB, as they are from the last instruction

being executed by the guest, the one that just triggered the EPT violation. We also note that if we need to access other different in-guest structures, not directly related to the last executed instruction, the processor might still need to perform a dedicated page walk and address translation, using the already available in-CPU page walking logic (just using a different root for the translation tree).
– ARM architecture processors [1] already include today control registers and instructions to perform translations from a GVA to either an intermediate physical address (GPA) or to a HPA.
– Regarding the proposed A/D-bit optimization, the CPU already knows during the EPT violation generation whether the fault was caused by the MMU page-walker setting the A/D-bit. This information is *already explicitly provided* in the exit-qualification field on EPT violations. Therefore, in our opinion, it would be simple and straightforward to just ignore the generation of the EPT violation for such cases.
– Intel delivered regularly new CPU enhancements during the last few years. Many of them are directly related to virtualization and security, such as EPT, `#VE`, nested-VMCS, very fast round-trip latency for VM-exits, APICv among others. From this, we conclude that they are actively looking into ways to enhance their platform. We can also easily realize that the complexity of any of those extensions is much greater than what we are proposing here.

4.4 Estimated Speed-up

Initially we considered extending BOCHS [2] to simulate the functionality of the proposed instructions, but abandoned the idea after we analyzed its capabilities. First of all, its slow simulation speed would mostly prohibit running any serious workload to be executed on. Secondly, such a simulation would not provide us any accurate and representative time measurements. Obviously, we did not had the capability to implement the instructions in real silicon either. Therefore, we decided to simulate the existence of these instructions by forcibly keeping some guest pages mapped inside the hypervisor virtual address space.

To estimate the speed-up of guest-memory access instructions, we performed a simple synthetic test: we created a test application that generated 1 million EPT violations in a loop by writing to a write-protected page. Each write generated an EPT violation, which in turn triggered a VM-exit. The MI logic analyzed each faulty attempt, but both the page containing the faulty instruction and the written page were kept permanently mapped inside the HVA memory space. Therefore, only the core processing was made by the MI, the guest memory reads being replaced by simple comparisons to validate the violation's address, and redirections to the permanently mapped HVA addresses. We believe this to be an as close as possible software estimation for what a real, in-silicon implementation could provide, as we replaced each READGVA with a sequence of only a few other x86 instructions. Besides, both the throughput and latency values of comparable MOV instructions are very low, so we believe that the estimation's replacement sequence to use *at least as much time* as it would take for any in-silicon implementation of those instructions to execute.

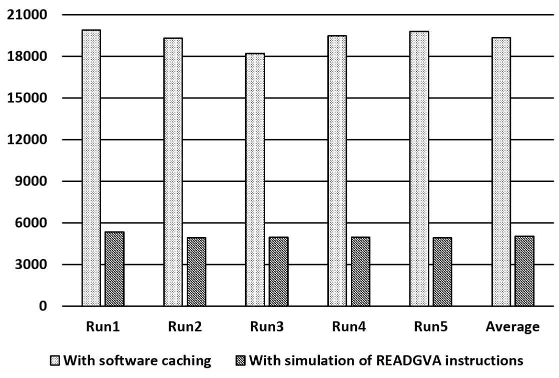


Fig. 11: EPT write violation speedup using the proposed new instructions (ms)

As seen in the results from Figure 11, with all our software caches, the total time needed to execute the test application was around 19.3 seconds. With the simulation of the fast instructions, this reduced to little over 5 seconds. While the performance boost is significant, it is important to point out that it would be infeasible to use this synthetic optimization on a larger scale. First of all, keeping guest pages mapped inside the hypervisor is not scalable because virtual addresses inside the guest may overlap with virtual addresses inside the hypervisor or in other guests, and they may be swapped in and out of the physical memory. Therefore, maintaining this kind of mappings can be done only for guest physical pages, and translation for guest virtual pages would still be needed. Even so, in the case of multi-guests, keeping these mappings synchronized would induce significant performance impact, not to mention the implementation complexity.

In the case of A/D-bit induced EPT violations we didn't need to do any further performance estimation. We argue that the performance impact would be eliminated entirely by our proposal, as no more EPT violations would be generated at all when the CPU updates the A/D bits. As we have shown already in Section 3.3, with an average processing time of about 10000 clock ticks inside the HV, the A/D-bit induced EPT violations processing may account for up to over 60% of the entire CPU time in intensive workload scenarios. We believe this to be a significant result to support in-hardware optimizations.

5 Related Work

There are several in-silicon improvements introduced by Intel in the last years that greatly enhance the performance of memory introspection. As a first example, Intel is steadily working with each CPU generation to reduce the round-trip VM-exit/VM-entry time, from over 3000 cycles in the original VT-x implementation to around 500 cycles in the Haswell [15] and around 400 cycles in their latest Broadwell [7] architecture. As another example, most likely, Intel is also working towards providing finer-grained memory monitoring CPU primitives [30]. Yet another technology is Virtualization Exceptions #VE, introduced in the Broadwell CPUs, which sustains moving the MI module inside a guest VM in order to further eliminate many of the round-trip overhead [12]. However, #VE has several limitations. Currently only EPT violations are delivered using this mechanism, and other events important for MI (e.g. such as MSR or CR accesses) need to be handled inside the hypervisor – thus forcing the implementation of split MI logic: some parts inside the HV, others in a guest agent. Besides this, in the context of a #VE agent, memory-validations are still needed – for example, the agent still has to translate a GVA that points inside user-space before accessing it. While not the scope of this paper, we can mention that our preliminary in-lab analysis indicated that using #VE to handle A/D bit update induced EPT violations can eliminate roughly about 25% of the overall overhead, compared with the results presented in Section 3.2. However it is important to point out, that such improvement would came mainly from the reduced round-trip latency of #VE and thus it would not be mutually exclusive with the A/D bit update speedup proposed by us, but instead, would lead to a cumulative improvement.

Memory introspection was introduced in [14] by Garfinkel and Rosenblum in 2003 as a way to implement an intrusion detection system that is well isolated from the host system. Jain et al. have done recently [17] a good survey of MI research. Jiang and Wang [18] implemented high-interaction honeypots based on memory introspection. Dinaburg et al. [9] uses virtualization to analyze malware. Among others, Dolan-Gavitt et al. [10] tries to overcome the semantic gap in automatic ways. Mohandas and Sahita [30] use virtualization and introspection to perform behavioral malware monitoring.

Originally VMware products [4] instrumented page table updates and other guest writes by the means of shadow page tables. However, on modern CPU with hardware accelerated SLAT, shadow paging mechanisms are not used anymore for efficiency reasons. Chang et al. [5] also describe a number of techniques to accelerate memory address translation, but their implementation is done in QEMU, based on binary translation, and thus is not applicable for our scenario. Somewhat similar to the #VE approach, Srinivasan et al. presents [33] a method to relocate the context of a monitored process so that it runs in the same context with the security agent (usually in a separate VM). This has significant implementation challenges on close-source operating systems. Sharif et al. proposes [32] the injection of an MI agent inside the monitored VM, to perform the most critical tasks. While protected by the hypervisor, any in-guest code can still be attacked through numerous vectors (e.g. such as via stack or shared data).

There are several noteworthy results on using MI in an asynchronous way, both for malware analysis [23, 24] and for on-premise security solutions [13]. We must however note, that our live VM protection scenario imposes very fast processing, thus we can't afford using tools like Volatility or relying on PDB metadata as Lengyel et al. does.

Vasudevan et al. performed a great work on identifying the requirements for trustworthy hypervisors on x86 platforms [36], and presented the implementation and formal verification of a module hypervisor framework [34]. They also present tamper-resistant execution environments for x86 platforms [35].

Well known, mainstream hypervisors, such as Xen, have been extended to support MI [25], and several traditional security solution vendors, such as McAfee/Intel [28] or Bitdefender [11], developed MI-based security technologies.

6 Conclusions

We analyzed some of the main performance overhead sources of EPT-level monitoring based synchronous live memory introspection, with focus on introspection of user-mode applications on x86 Windows platforms. Our research clearly indicates that a significant part of the overall overhead of user-mode introspection is induced by mappings of guest pages into the virtual memory space of the hypervisor. Our analysis shows, that even if we employ several software caches that reduce the mapping overhead by a factor of three, significant room remains for speedup. We believe the ultimate solution would be the introduction of new x86 instructions to allow reading/writing the guest virtual memory space from introspection logic executing inside the hypervisor's context. We strongly believe, that the proposed changes could fully eliminate both the memory mapping overhead and the overhead induced by numerous unnecessary VM-exits due to guest page table A/D bit updates incurred by memory introspection techniques. The overall speeding up of MI can be conservatively estimated to be at least 25%.

In the last few years Intel was continuously improving both the virtualization and security related in-silicon capabilities of the x86 processors, regularly introducing new instructions and technologies with every new processor generation. We argue that the proposed instructions could be rather easily incorporated into future x86 processors, as their functional complexity is much smaller than that of numerous recently introduced extensions and all required building blocks are known to be already present inside the processor.

Although simple in essence, the value and usability of the proposed extensions go far beyond the realms of MI based security. They could be used to speedup numerous other virtualization tasks, unrelated to security or introspection. Beside their presented form, the instructions themselves could be enhanced in several ways. For instance, they could be extended to support reading-writing memory not only inside the guest context of the active VMCS, but inside the guest context of an arbitrary VMCS, selected by a third operand.

References

1. ARM: ARM Architecture Reference Manual ARMv7-A and ARMv7-R (2014)
2. BOCHS: The cross-platform IA-32 emulator. `{http://bochs.sourceforge.net/}`, Accessed: 2014-11-24
3. BROMIUM: Bromium vSentry and LAVA products. `http://www.bromium.com/products.html`, Accessed: 2014-11-24

4. Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., Wang, E.Y.: Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. ACM Trans. Comput. Syst. 30(4), 12:1–12:51 (Nov 2012)
5. Chang, C.J., Wu, J.J., Hsu, W.C., Liu, P., Yew, P.C.: Efficient Memory Virtualization for Cross-ISA System Mode Emulation. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 117–128. VEE '14, ACM, New York, NY, USA (2014)
6. Chen, P.M., Noble, B.D.: When Virtual Is Better Than Real. In: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems. HOTOS '01, IEEE Computer Society, Washington, DC, USA (2001)
7. Chennupaty, Srinivas and Jiang, Hong and Sreenivas, Aditya: Technology Insight: Intel's Next Generation 14nm Microarchitecture for Client and Server (2014)
8. Citrix: XenClient XT. The ultimate in multi-level secure local virtual desktops. {http://www.citrix.com/products/xenclient/features/editions/xt.html}, Accessed: 2014-11-24
9. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis Via Hardware Virtualization Extensions. In: Proceedings of the 15th ACM conference on Computer and communications security. pp. 51–62. CCS '08, ACM, New York, NY, USA (2008)
10. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In: IEEE Symposium on Security and Privacy (SP). pp. 297–312. IEEE (May 2011)
11. Dontu, Mihai and Sahita, Ravi: Zero-Footprint Guest Memory Introspection from Xen. In: XenProject Developer Summit 2014
12. Durham, D.: Mitigating Exploits, Rootkits and Advanced Persistent Threats. In: Proceedings of the 2014 Symposium on High Performance Chips (Hot Chips 2014). IEEE Technical Committee on Microprocessors and Microcomputers in Cooperation with ACM SIGARCH (August 2014)
13. FireEye: Advantage FireEye. Debunking the Myth of Sandbox Security (2013)
14. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: In Proc. Network and Distributed Systems Security Symposium. pp. 191–206 (2003)
15. Hammarlund,Per: 4th Generation Intel Core Processor, codenamed Haswell. In: HotChips 2013
16. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual (January 2015), Accessed: 2015-02-02
17. Jain, B., Baig, M.B., Zhang, D., Porter, D.E., Sion, R.: SoK: Introspections on Trust and the Semantic Gap. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy. pp. 605–620. SP '14, IEEE Computer Society, Washington, DC, USA (2014)
18. Jiang, X., Wang, X.: "Out-of-the-Box" Monitoring of VM-based High-interaction Honeypots. In: Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection. pp. 198–218. RAID'07, Springer-Verlag, Berlin, Heidelberg (2007)
19. Joshi, A., King, S.T., Dunlap, G.W., Chen, P.M.: Detecting Past and Present Intrusions Through Vulnerability-specific Predicates. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. pp. 91–104. SOSP '05, ACM, New York, NY, USA (2005)
20. Lampson, B.: Accountability and freedom (Sep 2005)
21. Lampson, B.: Privacy and security: Usable security: How to get it. Commun. ACM 52(11), 25–27 (Nov 2009)

22. Laureano, M., Maziero, C., Jamhour, E.: Intrusion Detection in Virtual Machine Environments. In: Proceedings of the 30th EUROMICRO Conference. pp. 520–525. EUROMICRO '04, IEEE Computer Society, Washington, DC, USA (2004)
23. Lengyel, T., Kittel, T., Webster, G., Torrey, J.: Pitfalls of virtual machine introspection on modern hardware. In: 1st Workshop on Malware Memory Forensics (MMF) (Dec 2014)
24. Lengyel, T.K., Neumann, J., Maresca, S.: Virtual machine introspection in a hybrid honeypot architecture. In: Presented as part of the 5th Workshop on Cyber Security Experimentation and Test. USENIX, Berkeley, CA (2012)
25. LibVMI: Virtual machine introspection tools. {http://libvmi.com/}, Accessed: 2015-06-20
26. Ligh, M.H., Case, A., Levy, J., Walters, A.: The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. Wiley, 1 edn. (Jul 2014)
27. Luțaș, A., Lukács, S., Coleșa, A., Luțaș, D.: U-HIPE: hypervisor-based protection of user-mode processes in Windows. Journal of Computer Virology and Hacking Techniques pp. 1–14 (Feb 2015)
28. McAfee: A New Paradigm Shift: Comprehensive Security Beyond the Operating System (2012)
29. McAfee: McAfee DeepSAFE and Deep Defender (2013)
30. Mohandas, Rahul and Sahita, Ravi: Detecting Evasive Malware in Sandbox. In: Focus Security Conference 2014
31. Rutkowska, J., Wojtczuk, R.: Qubes OS. {http://www.qubes-os.org/}, Accessed: 2014-11-24
32. Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure in-VM monitoring using hardware virtualization. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 477–487. CCS '09, ACM, New York, NY, USA (2009)
33. Srinivasan, D., Wang, Z., Jiang, X., Xu, D.: Process Out-grafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. pp. 363–374. CCS '11, ACM, New York, NY, USA (2011)
34. Vasudevan, A., Chaki, S., Jia, L., McCune, J., Newsome, J., Datta, A.: Design, implementation and verification of an eXtensible and modular hypervisor framework. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. pp. 430–444. SP '13, IEEE Computer Society, Washington, DC, USA (2013)
35. Vasudevan, A., McCune, J., Newsome, J., Perrig, A., van Doorn, L.: CARMA: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security. pp. 48–49. ASIACCS '12, ACM, New York, NY, USA (2012)
36. Vasudevan, A., McCune, J., Qu, N., Van Doorn, L., Perrig, A.: Requirements for an Integrity-protected Hypervisor on the x86 Hardware Virtualized Architecture. In: Acquisti, A., Smith, S.W., Sadeghi, A.R. (eds.) Trust and Trustworthy Computing. pp. 141–165. TRUST'10, Springer-Verlag (2010)
37. Zhang, F., Chen, J., Chen, H., Zang, B.: CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 203–216. SOSP '11, ACM, New York, NY, USA (2011)