Exploiting the x86 Architecture to Derive Virtual Machine State Information

Jonas Pfoh, Christian Schneider, Claudia Eckert
Department of Computer Science
Technische Universität München
Munich, Germany
{*pfoh,schneidc,eckertc*}*@in.tum.de*

***Abstract*—Virtual machine introspection (VMI) describes the method of monitoring and analyzing the state of a virtual machine from the hypervisor level. Using knowledge of the virtual hardware architecture, it is possible to *derive* information about a guest operating system's state from the virtual machine state. We argue that by deriving this information it is possible to build VMI applications which are more robust against circumvention techniques than applications that do not rely on hardware knowledge. In this paper, we present various ways to leverage Intel's x86 architecture as well as the virtualization extensions from both Intel (VT-x) and AMD (SVM) to derive such information. Additionally, we describe how this derived information may be used in VMI-based security applications and against which threats they are most applicable.**

***Keywords*-Virtualization; Security; Anti-malware; Intrusion Detection; Introspection**

I. Introduction

Virtualization is a technology that allows one to run a guest operating system (OS) on a software layer, called the *virtual machine monitor* (VMM) or *hypervisor*, as if it were being run directly on the hardware. Virtualization clearly has its applications in the field of IT security. For example, it enables the analysis and manipulation of the state of a guest operating system running inside the virtual machine (VM) from the isolation of the hypervisor. This method of combining isolation, inspection and interposition is referred to as *virtual machine introspection* (VMI) [1].

The most crucial part of any VMI application is the extraction of appropriate system state information from the binary data that comprises the virtual machine state. This process is called *view generation* [2]. Any information that is not extracted from this binary data will not be available for further processing. Therefore the view generation must retain all information relevant for a given application. It is also important to perform this view generation in the most robust manner possible, that is, it is ideally immune to any malicious influence.

Pfoh et al. [2] describe three fundamental patterns for view generation (ref. Section II). From these three patterns, the *derivation pattern* is the most robust as it only makes use of knowledge about the virtual hardware architecture in order to generate the view. More precisely, from the perspective of the virtual machine, the state information is collected at the lowest level possible. There is no way for an attacker inside the virtual machine to perform actions at a level lower than the hardware in order to circumvent introspection and hide his activities from the hypervisor.[1]

In this paper, we investigate the potential of Intel's x86 architecture and the virtualization extensions from both Intel (VT-x) and AMD (SVM) for derivative view-generation. We identify and describe methods for extracting security relevant information about a guest system from its low-level state through knowledge of the hardware. These methods allow one to enumerate the running processes, monitor system calls, or track network connections on a per-process basis in a completely guest OS independent manner. Some of these methods may be improved by adding knowledge of the guest operating system. With this, for example, the entire system call mechanism can be monitored in such a way that it cannot be circumvented by any malicious activity within the guest.

The remainder of this paper is organized as follows. In Section II we introduce some terminology and briefly describe view-generating patterns. Section III motivates our work by outlining the advantages of the derivation pattern. The possible threats that we intend to counter follow in Section IV. In Section V, the actual mechanisms for derivative approaches are presented and discussed. Finally, Section VI concludes this work.

II. Background

In this section, we introduce some terms and concepts which were established by Pfoh, et al. [2]. This will help for a better understanding and discussion of the motivation in Section III and the hardware-based mechanisms for view generation in Section V. Since we only cover the required minimum and do not detail every definition, we encourage the reader to consult the aforementioned publication for an in-depth discussion of the foundations we are using in this work.

VMI is a technique for inspecting and manipulating the state of a guest OS from the isolation provided by the hypervisor. The hypervisor has a complete and untainted

[1]For our considerations, we make the idealistic assumption that an attacker cannot break out of the virtual machine. However, we are well aware of the fact that privilege escalation attacks for several widely-spread hypervisors do exist.

978-0-7695-4095-5/10 $26.00 © 2010 IEEE
DOI 10.1109/SECURWARE.2010.35

IEEE computer society

view of all guest system state as well as the ability to manipulate that state. These are ideal properties for security applications.

The system state of a virtual machine can be described as the combination of all CPU registers, all volatile memory, the entire contents of stable storage, virtual BIOS settings, and so on. The challenge is that the hypervisor has no inherent knowledge of the meaning of this state. That is, a particular memory region or register may be used differently among different OSs. The hypervisor has no inherent knowledge of the role a particular piece of the guest system state plays. Chen and Noble refer to this lack of knowledge as the *semantic gap* [3]. The set of knowledge the hypervisor requires in order to determine the role of each piece of guest system state is referred to as *semantic knowledge*.

In summary, the state alone is not very useful until some semantic knowledge is applied to it. For example, the contents of kernel memory is only useful once we know where key data-structures and functions lie within that memory. The representation of the VM state after some semantic knowledge has been applied is called a *view* of the VM. In our example, the kernel memory is a portion of the *guest system state*; *semantic knowledge* is applied to this state in the form of key data-structure and function locations; and finally, these key data-structures and functions are extracted as the *view*.

There are several ways for the hypervisor to bridge the semantic gap. Pfoh et al. [2], present three fundamental patterns for bridging the semantic gap, namely: the *out-of-band delivery* pattern, the *in-band delivery* pattern, and the *derivation* pattern. They differ in two ways, first, where within the architecture the view generation takes place (i. e., internally or externally with respect to the VM) and second, how semantic knowledge is incorporated. Additionally, we found that every published VMI approach we are aware of can be described as a combination of one or more of these patterns. We will briefly summarize these patterns in the following.

A. Delivery Patterns

The in-band and out-of-band delivery patterns are the most widely used methods for bridging the semantic gap in various VMI approaches. Here semantic knowledge is *delivered* to the external view-generating component. This can either happen *out-of-band* (i. e., before VMI begins), or *in-band* (i. e., at run-time).

For example, in the out-of-band delivery pattern the view-generation might rely on a previously delivered symbol table based on the guest OS kernel to determine the position and layout of kernel-level data-structures in memory. While, in the in-band delivery pattern the view-generation might rely on a daemon running inside the VM which provides the view-generation with information about all active processes.

B. Derivation Pattern

The derivation pattern does not require any knowledge about the guest operating system or any other software component but rather *derives* information through semantic knowledge of the virtual hardware architecture. For example, understanding the function of particular control registers within the virtual CPU and monitoring their contents provides information about the current state of the system. It is this pattern that we will focus on for the remainder of this paper.

III. Motivation

VMI is a strong enabler for derivative methods. However, many published VMI-based approaches [1], [4], [5] fail to consider VMI as a new paradigm, thus failing to acknowledge the potential advantage of derivative methods. As an example, if one is monitoring a variable within a software system for change, it is possible to circumvent this monitor by altering the underlying system in such a way that the monitored variable is no longer used. On other hand, if one is monitoring a critical hardware register, it is impossible for any malicious entity to alter the system such that the register is no longer used. In this example, monitoring the hardware register is totally robust against circumvention because the hardware interface cannot be changed. In the same way, derivative approaches are more robust against circumvention than the more common delivery approaches.

As it turns out, the amount of information we can gather about a system based solely on knowledge of the hardware is limited. However, it is possible to combine a derivative approach with delivered knowledge in such a way that we retain this robustness against circumvention. This requires rooting delivered knowledge in a derivative component we will call the anchor.

To illustrate this, we consider a component of Garfinkel and Rosenblum's Livewire system [1]. We choose this system because it is well known. Their polling policy modules monitor low-level system state which is resident in memory. This polling module may be used to monitor the integrity of critical kernel data-structures, such as the system call table. However, this module would be susceptible to a class of attacks that manipulate the Interrupt Descriptor Table Register (IDTR) in order to circumvent use of the monitored system call table all together. Since this module only considers state resident in memory, such an attack is not detected. The IDTR is a system register that holds the base address of the Interrupt Descriptor Table (IDT). This mechanism is explained in more detail in Section IV-A.

We would suggest augmenting this module to not only consider memory, but also CPU registers, thus allowing one to anchor delivered knowledge (location of the system call table) in a derivative component (IDTR monitor). The position of the system-call table in memory can be retrieved by following the address in the IDTR, the address of the corresponding interrupt dispatcher in the interrupt description table (IDT), and finally the actual dispatching code as shown in Figure 1. Here the robustness against circumvention is increased in that any change to the system call table will be detected nor can the system be altered in such a way that the system call table is not used.

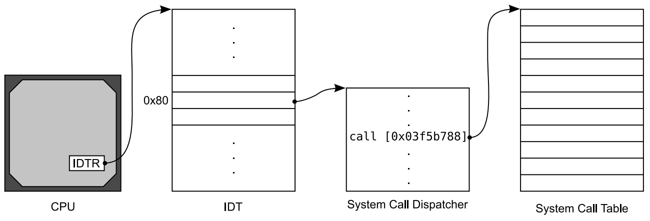

Figure 1. Rooting the system-call table to a hardware register of the virtual CPU

While the above example is fairly trivial, this same approach can be applied in far more complex scenarios. Our contribution in this paper is a thorough investigation of the x86 architecture in order to determine what potential this architecture has with respect to derivative methods for VMI. We present several building blocks which may be used to anchor security mechanisms in the hardware of the system.

IV. Threats

There are certain threats to system security which can be countered by mechanisms that would benefit from a hardware anchor. These threats are presented here. The building blocks that provide the hardware anchor through derivative methods are then presented in Section V.

A. Interrupt Hooks

Setting interrupt hooks is a common tactic among rootkit authors, especially hooks in the system call mechanism. This allows the rootkit to perform keystroke logging, for example, to ascertain further passwords for the intruder. To further understand this threat, it is important to understand the interrupt mechanism as it is implemented in the x86 architecture.

The standard interrupt mechanism makes use of a system register, the IDTR. This system register contains the address of the Interrupt Descriptor Table (IDT), which, in turn, holds the addresses of the various interrupt handlers. When an interrupt occurs, the system refers to the IDT to determine the address of the appropriate handler, then invokes this handler.

A system call is a software interrupt, though the mechanism is a bit more complex as there are two primary ways a system may choose to handle them. The first is through the `INT` instruction which triggers a software interrupt. The second uses dedicated `SYSENTER` or `SYSCALL` instructions. Further technical differences are irrelevant in this section, but note that each of the mechanisms makes use of specific system registers that point either to the IDT or to the system call dispatcher (i. e., an interrupt handler) directly. Modern operating systems generally support both mechanisms.

Another critical component of the system call mechanism is the system call table. This is generally a kernel data-structure that contains the entry point addresses for the individual system call handlers. That is, the system call dispatcher consults the system call table to determine the memory address of the respective handler, then invokes that handler.

Considering these two ways for system call invocation, there are four possible stages at which an attacker could possibly perform system call hooking:

1) Manipulate the appropriate system registers and/or IDT to point to a malicious system call dispatcher.
2) Manipulate the system call dispatcher to consult a malicious system call table.
3) Manipulate the system call table to point to malicious handler routines.
4) Manipulate the system call handlers themselves to perform malicious activity.

In order to protect against interrupt hooking, verifying the integrity of the interrupt mechanism at *all* possible stages is imperative. That is, one must protect the integrity of the system registers, the IDT, the interrupt handlers, the system call table, and the system call handlers.

B. Information Hiding

Hiding information from the guest OS is another common tactic employed by rootkits. Information hiding may include a wide range of goals, but we will focus on three of the most common, namely: process hiding, kernel module or driver hiding, and network activity hiding. All of these may be used by an attacker to disguise his presence on a compromised machine. The issue with such hiding is that, from the perspective of an attacker, there exist several ways to reach each goal. However, there are commonalities which can be exploited by a security application to counter each goal. These are outlined below.

1) Process Hiding: Process hiding is the act of hiding a process or task from tools within the operating system. A competent user or system administrator will expect to see certain processes and an unknown process may cause suspicion. An attacker will want to avoid any such suspicion and therefore employs process hiding techniques.

One way to counter process hiding is to perform an enumeration of running processes (tasks) at two different levels. First, a "low-level" task enumeration is performed. That is, the number of unique processes in an OS is determined at a level lower than the dedicated OS interface provides. Then a "high-level" enumeration is performed by using the OS interface (e. g., using `ps -A` in Linux). The collected information can then be compared and any inconsistencies indicate that process hiding is taking place. For such a technique to work the low-level task enumeration must be performed at a level lower than the level at which the process hiding is taking place.

If the low-level task enumeration is performed at the lowest level possible, then a malicious entity does not have the opportunity to circumvent the enumeration. This lowest possible level is the (virtual) hardware. That is, no matter how sophisticated a process hiding technique may be, it cannot circumvent the fact that even hidden processes must get scheduled on a CPU in order to carry out their payload. When a process with its own virtual address space is scheduled, the process's unique set of page directories and tables must be loaded and made available to the CPU. That is, these page directories and tables cannot be hidden due to hardware specifications. This leads to several methods for low-level task enumeration, some of which have already been used in previous work [6], [7], [8], [9].

2) Kernel Module and System Driver Hiding: As with process hiding, hiding kernel modules or system drivers is the act of hiding these kernel-level components from tools within the operating system and thus from the user.

While the goal is similar to that of process hiding, the difference is that these components get loaded into kernel memory and do not possess a separate virtual address space. To complicate things further, these components may continue to perform normally even when the kernel's handle to the particular component has been removed. That is, the kernel is completely unaware of such a component and yet it continues to perform its tasks. For example, a Linux module may be loaded into memory, making the code resident in kernel space. A malicious entity may then remove the entry for this module from the module list without unloading the code. This results in malicious code being resident in the kernel without any indication that it is present. [10]

Of course, the memory which is in use by a hidden component must be allocated and one could take the approach of monitoring all allocated portions of kernel memory and identifying all unknown components (i. e., components for which the kernel has no handle). However, this approach is highly cumbersome and impractical as it requires semantic information about every piece of kernel information in order to determine if a hidden components exists.

An alternative approach is based on the observation that such a component must get loaded before it becomes hidden and the loading of such components must be handled within the kernel at a specific location, often through a system call. That is, by monitoring the appropriate system call for such components we have the opportunity to recognize when kernel modules or system drivers are loaded and thus alleviate the need for detecting hidden components. We may use the same method described in Section IV-B1, comparing this VMI view to an in-OS view. It is important to note that this method relies heavily on the guest OS. If the guest OS provides methods other than system calls to load kernel code, an approach anchored in a derivative component may not be possible.

3) Network Activity Hiding: Another common practice of rootkits is to hide network activity. That is, an attacker will want to hide all network traffic created by his tools from the OS as this is another common avenue by which users and administrators become suspicious. For example, an attacker may want to set up a back-door that listens on a particular port. The attacker will want to hide the fact that a service is listening as well as any traffic created by this service.

Any traffic created within a VM must travel through the hypervisor as the hypervisor manages all virtual I/O devices that are extended to the VM.[2] This can be taken advantage of and in fact it is impossible for any changes within the guest to hide network activity (or any I/O activity) from the hypervisor. In addition, it is possible to correlate network packets with processes on the system, as we will explain in Section V-A4.

4) Virtual Machine Rootkits: While not seen in the wild, rootkits that move the currently running OS into a VM have been shown as a proof of concept [11]. Such rootkits make use of the `VMENTRY` instructions provided by the virtualization extensions of modern CPUs. Such rootkits can be thwarted by disallowing the use of these `VMENTRY` instructions from within the guest OS or trapping and regulating their usage.

C. Network-Based Attacks

Network-based attacks include any form of attack or malicious behavior which is carried out over a network. Ideally these attacks are recognizable when monitoring network traffic.

Traditional Network Intrusion Detection Systems (NIDS) are a common tool in any security solution. These systems listen on a network wire and report traffic that is deemed suspicious, usually based on a set of signatures, i. e., patterns of malicious network traffic. As mentioned in Section IV-B3, all network activity from a guest OS is visible to the hypervisor, thus introducing a NIDS within the hypervisor is a fairly trivial task. This NIDS can be used to protect the guest as well as to detect misbehaving processes in the guest OS. Furthermore, information from the NIDS can then be combined with contextual information about the process sending or receiving the packet to raise its accuracy.

With this contextual information it is possible to provide connection profiles for individual processes. That is, we can correlate outgoing packets with unique processes, thus creating a network activity profile on a process granularity. These profiles can then be used to detect patterns or match signatures at a per-process level.

[2]There do exist exceptions to this in which devices are passed to the VM at a PCI level, but since these exceptions can be explicitly avoided in cases where VMI is the goal, we disregard them.

D. Malicious Processes

Malicious process detection is a fairly broad goal and recognizing the subtleties of all malicious processes is unlikely if not impossible through VMI. However, there are some areas in which VMI can be used to support malicious process detection.

As mentioned in Section IV-C, connection tracking may be one avenue for malicious process detection. In addition to building and analyzing a network profile, it is possible to detect malicious or misbehaving processes based on system call traces. Once we have a system call profile for a process, this information can be used to identify misbehaving processes through uncommon activities. Such techniques have been presented using system call traces collected from within the monitored system. [12], [13] By using derivative methods the properties of these approaches could be strengthened.

V. Hardware Mechanisms

In this section, we outline some of the hardware mechanisms that become the building blocks of a derivative approach or a combination approach that is rooted in hardware. We look at the x86 architecture from two perspectives. First, we explore the introspective features that the architecture provides when used as virtual hardware. These features include primarily passive information gathering techniques. Second, we show how the virtualization extensions can be exploited for introspection purposes. As we will see, they also allow us to actively intercept certain events and instructions triggered inside the guest.

Information presented in this section is based on specifications found in the Intel and AMD Developer Manuals [14], [15] unless otherwise noted. We reinforced this research through experimentation using the Linux Kernel Virtual Machine (KVM) [16].

A. VM State Access

A very straightforward mechanism for VMI is simply a matter of checking and comparing CPU and/or memory state at regular intervals. The challenge in such an approach is determining which portions of the state are helpful. This section outlines exactly those portions of the state based on knowledge of the x86 architecture.

In addition to identifying those portions of state which are helpful, one must consider the trigger upon which this state is inspected. A timer may be the simplest solution. Although, event-based triggers may often be more appropriate. For example, if we wanted to use a context switch as the event trigger, we may use the mechanism suggested in Section V-B1.

1) Control Register 3 (CR3): The CR3 register holds the address of the top-level page directory for the current context when paging is activated. That is, each process has a unique top-level page directory and the address of this page directory is stored in the CR3 register. Since each process has a *unique* top-level page directory, the address of this directory acts a unique identifier. Hence, monitoring this register may be used to enumerate processes. The challenge here is determining when a process is in fact no longer running as opposed to simply idle. One may have to rely on heuristics and create a baseline for the running system to determine a reasonable idle time. The Lycosid system [8], [9] makes use of a exactly this mechanism to enumerate processes within a system.

2) IDTR and System Call MSRs: Monitoring the IDTR and the system call MSRs (machine specific registers) allow us to monitor for the placement of interrupt hooks at this level. Generally such a hook is placed within the system call mechanism, though hooks could be placed to catch any interrupt within the system. We will focus on monitoring for system call hooks, however the process described here for hooking system calls based on interrupts may be used to monitor for any hooks within the interrupt system.

As mentioned in Section IV-A, there exist two primary system call mechanisms for system calls within the x86 architecture. The first method is to use the IDT to store the address of the system call handler and use the `INT` instruction to perform the actual system call. The base address and size of the IDT are stored in the IDTR and the layout of the entries in the IDT (gate descriptors) are specified by the hardware architecture. These gate descriptors provide a logical address for each interrupt handler. This information allows us to monitor for any interrupt hooks (including system call hooks) at the IDTR and IDT level.

The second, more recent method involves MSRs which are responsible for handling fast switching to the system call handler. This fast system call mechanism is provided through the `SYSENTER` or `SYSCALL` instructions. Going forth we will refer to the Intel syntax (i. e., `SYSENTER`) as the technical differences of these two mechanisms are irrelevant here. The MSRs store, among other things, the offset for the system call handler.

In the case of system calls, we refer to the code that the IDT or the appropriate MSR point to as the *dispatcher* routine. We give it this distinction because this function does not “handle” the system call. Rather, it examines the calling conditions and refers to a data-structure called the *system call table* in order to determine where the appropriate handling function is located, then hands the appropriate information over to this handler.

As mentioned in Section IV-A, monitoring for hooks at the register or IDT level alone is not sufficient. We must monitor the integrity of all stages of the system call mechanism (c. f. Section IV-A), i. e., the appropriate registers, the IDT, the system call dispatcher, the system call table, and the system call handlers. Since the location and layout of some of these functions and data-structures are not dependent on the hardware architecture, some delivered knowledge will be necessary.

It is possible to combine such a derivative method with a delivery approach in which the size and layout of the dispatcher and handler functions and the system call table are delivered to the hypervisor. Here we have an example of rooting the semantic information in the hardware as described in Section III, thus providing a solution with increased robustness against circumvention. Let us consider a simple example. We have the ability to monitor the IDTR and therefore the IDT. The address of our dispatcher lies within the IDT and the address of the system call table lies within that dispatching procedure. Finally, the addresses of the handlers are contained within the system call table. Even though we rely on delivered information (i. e., the location and layout of the dispatcher, handlers, and system call table), it is impossible for a malicious entity to manipulate anything along this chain without the change being evident from the hypervisor since this chain is rooted in the hardware (IDTR or MSRs). If the malicious entity alters the system call table, the change is detectable. If it manipulates the dispatcher to use a cloned system call table, the change is also detectable. This chain continues until we reach the hardware (in this case the IDTR) and any changes here are clearly detectable as well. At this point an attacker cannot resort to a level lower than the hardware.

Both Windows and Linux use system call mechanisms similar to the one described here. However, it is possible to execute the kernel in a separate virtual address space rather than in a dedicated region of other process's address space. In this case the mechanism would look slightly different, though a similar technique could be incorporated.

3) GDTR and LDTR: The Global Descriptor Table (GDT) and Local Descriptor Table (LDT) are data-structures that are crucial to the segmentation mechanism in the x86 architecture and the GDTR and the LDTR are the registers that contain the location and size of GDT and LDT, respectively. The layout of these tables is specified by the hardware specifications and they may be used to regulate the segments that are accessible at various protection levels and facilitate the switch between these protection levels. Since these data-structures play a part in the protection mechanisms of the x86 architecture, they may become a target for a malicious entity. Thus, monitoring them and their respective data-structures for integrity may become crucial in a security solution.

In practice, monitoring these registers and their respective data-structures has limited usage as OSs may (and often do) choose to effectively ignore this mechanism. That is, the OS may choose to provide isolation and protection by leveraging the paging mechanism and use the segmentation mechanism in a minimal way. For example, the Linux 2.6 kernel sets up all kernel and user segments such that the logical addresses are equivalent with linear addresses. This means that both protection levels share the same segments and this mechanism provides no protection at all. This leaves little for a malicious entity to abuse in this respect. Though, in systems which rely on segmentation for protection monitoring the integrity of these registers becomes crucial.

4) Virtual I/O: The VM must rely on the hypervisor for all input and output since the guest OS is provided virtual I/O devices by the hypervisor so that the physical device may be shared by multiple VMs or by the hypervisor itself. This can, of course, be leveraged by VMI mechanisms to monitor network traffic or keystrokes.

Performing such I/O monitoring is a matter of tapping into the virtual I/O device within the hypervisor and interpreting the data. When keystroke logging is the goal, for example, this would be a matter of tapping into the appropriate component of the hypervisor and logging the data stream to a file. Network traffic monitoring is another interesting application of this mechanism.

A simple example of taking advantage of network traffic monitoring is to perform network-based intrusion detection or firewall duties from the hypervisor. However, more interesting is to leverage the fact that the mechanism is performing its duties from the hypervisor. For example, Srivastava and Giffin [17] leverage this fact to present an application-aware firewall which is isolated from the running guest. Such an approach requires some delivered information, but is an excellent example of tapping into a virtual I/O device and combining this approach to come to a result that brings together the primary advantages of both host-based (i. e., application-awareness) and network-based (i. e., isolation) firewalls.

It is possible to take this one step further by combining the ability to tap into the network stream with a simple virtual CPU state inspection of the CR3 register. This register may be used to identify unique processes running in the VM as discussed in Section V-A1. This allows us to create network activity profiles for unique processes and would allow us to perform process-specific intrusion detection and firewall duties. That is, if a process is tagged as acting suspicious it may be blocked on a process granularity. This approach maintains isolation and relies completely on derived information and is therefore robust against circumvention techniques that might try to hinder application-aware solutions from correlating a network stream to an application correctly.

B. Virtualization Extensions

This section goes into depth considering how to best leverage several aspects of the virtualization extensions provided by both Intel (VT-x) and AMD (SVM) for VMI. Unless otherwise specified, these techniques are possible on Intel as well as AMD hardware which provide similar x86 virtualization extensions.

1) Context Switch Trapping: The virtualization extensions allow one to directly trap to the hypervisor on a context switch. This results in a fairly straightforward method for trapping context switches, however this method relies on the guest OS's usage of the TSS. It is possible for an OS to use the TSS mechanism in a limited way and perform software context switches. For example, the Linux 2.6 kernel makes limited use of the TSS, having a single TSS for each CPU. This results in no hardware context switches, making this technique useless.

Since the above technique is not useful for all guest OSs, we discuss another technique that does not have the above restrictions and is also supported by both sets of CPU extensions. This technique traps writes to the CR3 register. The CR3 register is the control register which holds the offset of the current page directory as described in Section V-A1. Since each process must have its own page directory in OSs that make use of hardware-supported virtual memory, trapping writes to the CR3 register will result in trap to the hypervisor on each context switch.

In fact, many hypervisors that use shadow page tables must trap all context switches (i. e., CR3 accesses) in order for the VM to run properly. In this case the work of trapping is already done, one must simply incorporate their intended action.

2) Virtual Machine Entry Trapping: As mentioned in Section IV-B4, proof of concept malware does exist that demonstrates the ability to move the running OS into a VM, thus allowing the malware to perform introspection and take advantage of interposition [11]. In order to do this, a piece of malware must take advantage of the hardware extensions for virtualization. One must simply disallow any attempt to enter into a VM and react to this (e. g., logging, setting an alert, etc). Both Intel and AMD provide the ability to trap the entry instruction, so this can be done with relative ease.

3) System Interrupt Trapping: System-specified interrupts are those interrupts which reside within the IDT at offsets from 0 to 31. Common system-specified interrupts are page-fault exceptions and general protection faults. For a complete list we refer the reader to the Intel Developer Manuals [14]. It is possible to set a VM to trap to the hypervisor on any of these interrupts. That is, any action which causes a system-specified interrupt may be trapped by the hypervisor.

This can be exploited by security applications in some cases by manipulating the guest from the hypervisor so that an event artificially causes a system-specified interrupt which is then trapped by the hypervisor. Such approaches are discussed in Section V-C.

4) User Interrupt Trapping: User defined interrupts use the same mechanism as system defined interrupts, though they must use an interrupt code within the range 32...255 (this also acts the offset into the IDT) and are always caused by using the `INT` instruction. For example, system calls may be implemented in this way and thus may be trapped. The virtualization extensions provided by AMD allow native trapping of user defined interrupts in the same way that system defined interrupts are trapped.

In order to trap these interrupts on Intel machines some extra steps must be taken. For a full discussion on this, see Section V-C3.

C. System Interrupt-enabled Mechanisms

As mentioned in Section V-B3, it is possible to leverage the fact that the x86 virtualization extensions allow us to cause the guest OS to trap to the hypervisor due to a system-specified interrupt. Often times we may want to cause the VM to exit due to an event for which there exists no mechanism to cause a `VMEXIT` (i. e., trap to the hypervisor). In this case, it may be possible to manipulate the VM state in such a way that the target event will cause a system-specified interrupt and set the hypervisor to trap this interrupt. The challenge here is how to manipulate the VM state so that the target event causes a trappable interrupt and how to minimize false positives. In this context, *false positives* are traps to the hypervisor that are due to our manipulation, but do not contribute to VMI and *legitimate* interrupts or exceptions are interrupts or exceptions which would have occurred regardless of our manipulation.

Many false positives will lead to significant performance overhead. It is therefore necessary to carefully craft such a mechanism in order to reduce the number of false positives. In addition, involving the hypervisor in legitimate interrupts where the hypervisor intervention is not necessary will also lead to unwanted performance overhead. Both of these performance factors must be taken into account when considering such an approach.

The final challenge in such a mechanism is differentiating between legitimate interrupts and those caused by our manipulation of the VM state. This step will vary with each approach and is discussed in the appropriate subsections below.

1) Avoiding Countermeasures: Changing the state of the VM leads to the possibility that an entity within the guest OS can detect or even deter our mechanism by looking for these changes and possibly altering this state. For this reason it is imperative to monitor the integrity of the corresponding state from the hypervisor. In the simplest case this can be achieved by monitoring the state at regular intervals. It may also be possible to use paging protection mechanisms in a hypervisor that makes use of shadow page tables since the page tables which are actually being used reside outside of the VM. Finally, in some cases it may be possible to deter a malicious entity from detecting the mechanism at all by trapping access to this state. For example, the x86 virtualization extensions allow us to trap accesses to most system registers and MSRs.

2) Page-Fault Exception: A page fault is an exception that is produced by the hardware when an exception occurs within the paging system of the architecture. We will concentrate on three specific causes of page faults which are helpful for VMI:

1) A page is requested for which the page-present flag is not set. We will refer to this as the Page Not Present (PNP) exception.
2) An attempt to write to a page for which the write flag is not set. We will refer to this as the Page Illegal Write (PIW) exception.
3) An attempt to execute operations on a page for which the non-executable flag is set. We will refer to this as the Page Illegal Execute (PIE) exception.

The PNP exception may be leveraged to indicate any type of access to a particular page belonging to a process. That is, the hypervisor may unset the page-present flag for a specific page whether the page is present or not and cause all page faults to trap to the hypervisor. The hypervisor must then intercept all page fault exceptions which were caused due to a PNP exception and for which the causing page is the page in question. The Ether system [18] uses such an approach to monitor system call activity. Rather than unsetting the page-present flag associated with the page in which the system call dispatcher is resident, it replaces the data in the MSR which holds the offset of the system call dispatcher with an offset into a page which is always set to be not present. This is a clever use of this exception as it allows the authors to trap all system calls using this system call mechanism, without having to consider false positives that would be caused by normal execution within the page in which the system call dispatcher is resident.

The PIW exception may be leveraged to indicate attempts to overwrite portions of memory. To make use of this the hypervisor must unset the write flags for the appropriate pages and handle these exceptions. This is a straightforward approach and works well when write protecting static data-structures or code segments. In addition, the PIE exception may be leveraged to indicate attempts to execute certain portions of memory. These are straightforward methods as we are simply using the mechanism in the way they were intended although we are doing so from outside of the guest OS. Litty et al. [6], [7] make use of the PIE exception in yet another way. They use this mechanism to perform task enumeration. That is, they set the pages of a process to non-executable to determine whether the process is still running.

Changes made to force these interrupts are not visible to the guest OS in a system which uses shadow page tables as the shadow page tables are resident outside of the VM. This makes for a method which is robust against detection and manipulation attempts from within the guest VM.

This approach also seems as though it could be leveraged to perform code execution trapping. That is, trapping to the hypervisor when a portion of memory legitimately containing code is executed. This may be done to monitor the entrance into a function, for example. However, we caution the reader when using this approach for the following reasons: Such a method for code execution trapping will cause a trap on *each* instruction on the page causing many false positives; but more problematic, it forces the hypervisor to emulate all these instructions. This could solved by unsetting the non-executable flag after the first instruction is executed, though this leads us to two further problems. First, the hypervisor needs to continuously poll the VM state to determine whether it can reset the non-executable flag or we must set a time in which we are sure the function has completed execution. The second problem is that in this time another process could enter the same function without the hypervisor being made aware. Given these potential problems, the paging mechanism is not the proper approach for code execution trapping. For viable methods see Sections V-C5 and V-C4.

3) General Protection Exception: The general protection exception is produced for a wide variety of specific reasons all having to do with privilege protection. For the purposes of VMI we concentrate on the following five specific reasons:

1) An attempt to execute operation within a segment that is not executable. We will refer to this as the Segment Illegal Execution (SIE) Exception.
2) An attempt to write to a read-only data segment. We will refer to this as the Segment Illegal Write (SIW) Exception.
3) An attempt to read from a execute-only code segment. We will refer to this as the Segment Illegal Read (SIR) Exception.
4) Referencing an illegal or non-existent entry within the IDT. We will refer to this as the Illegal IDT Reference (IIR) Exception.
5) Loading the CS register with a null segment selector. We will refer to this as the CS Null Segment Selector (CNSS) Exception.

Leveraging the first two exceptions, namely SIE and SIW, works almost identically to the mechanism described in Section V-C2 with regard to the PIE and PIW exceptions. The only difference is that here we are working within the segmentation mechanism of the hardware rather than the paging mechanism. This leads to a difference in granularity as the size of a segment is not static like the size of a page, though the VMI mechanism is almost identical except that here we set the hypervisor to trap and interpret a different exception.

The third cause for an exception is also due to the segmentation mechanism of the hardware. The VMI mechanism to leverage SIR exceptions is identical to that for the SIE and SIW exceptions, though here we have the ability to do something that the paging mechanism does not allow, namely trap attempted read access to code segments. This is useful for avoiding detection of VMI mechanisms that change code, as described in Section V-C4.

The issue with any VMI mechanism that leverages the hardware segmentation mechanism is that segmentation may not be properly used by modern OSs that use paging, as discussed in Section V-A3. Linux kernel 2.6, for example, uses this mechanism very minimally. So while these mechanisms may be useful in the correct environment, they rely on the guest OS's usage of segmentation.

The fourth cause for a general protection exception is the IIR exception. Unlike the above general protection exceptions, the IIR exception is independent of segmentation. This exception is useful for trapping user specified interrupts on Intel architectures. As explained in Section V-B4, the Intel virtualization extensions do not provide a mechanism for trapping interrupts greater than 31 (i. e., user specified interrupts). As with the previously described methods, the hypervisor must be prepared to trap this particular exception. In addition, the hypervisor must alter the IDT entries for the interrupts that it wishes to intercept. This is simply a matter of invalidating the entry by unsetting the segment present flag in the call gate. Now any reference to this entry will result in a general protection exception which can be trapped by the hypervisor. Similarly, the hypervisor could set the Descriptor Privilege Level (DPL) for each IDT gate descriptor to 0. This would result in a general protection exception each time a process with a Current Privilege Level (CPL) greater than 0 (a user process) tries to access the gate.

Finally, the CNSS may be used in a very specific way to facilitate system call trapping when the `SYSENTER` instruction is being used. Among other things, the `SYSENTER` instruction will set the CS register based on the contents of SYSENTER_CS_MSR register. That is, saving the contents of the SYSENTER_CS_MSR in the hypervisor and setting it to null will result in an CNSS exception any time the `SYSENTER` instruction is executed. This exception can be trapped to the hypervisor and the instruction emulated, thus, allowing one to trap all system calls if the system uses the `SYSENTER` mechanism.

4) Invalid Opcode Exception: The invalid opcode exception provides a mechanism that allows the kernel to handle invalid opcodes should the CPU attempt to execute them. As with all the mechanisms described in Section V-C, this mechanism relies on the hypervisor being set to catch and handle guest exceptions.

This mechanism lends itself to performing code execution trapping. In order to do this, one must simply replace the opcode at the position one wishes to trap with an invalid opcode. Then, the hypervisor can distinguish this from legitimate invalid opcode exceptions by referring to the opcode that was intended to be executed. This does, however, require that the hypervisor emulate the replaced operation, which may be a challenge.

This is straightforward, though if a malicious entity where to know where to look, such a mechanism is easy to detect and even circumvent if further measures are not taken. For this reason, such an approach *must* be combined with a mechanism to write protect the appropriate area of memory in order to deter circumvention such as those described in Sections V-C2 and V-C3.

This exception has additional usefulness that is closely related to the CNSS exception discussed in Section V-C3. If the use of the `SYSCALL` instruction is deactivated in the Extended Feature Enable Register (EFER)—this is a matter of unsetting a flag within the EFER—and the instruction is used, an invalid opcode exception is raised. As with the CNSS exception, this may be used to facilitate system call trapping, except that this method is useful if the system uses the `SYSCALL` instruction to perform system calls. The EFER must be manipulated from the hypervisor, making the `SYSCALL` instruction invalid. Then, the invalid opcode exception must be trapped to the hypervisor, the `SYSCALL` instruction emulated, and control returned to the guest.

5) Debug Exception: The x86 architecture provides debugging support through the use of debug registers and a Debug Exception. As part of this support the architecture allows one to define four linear addresses as addresses whose execution cause a debug exception. This exception can then be set to trap to the hypervisor. This gives us the opportunity to set four memory addresses whose execution will be trapped to the hypervisor. In addition, accesses to these registers can be trapped so that any attempt to change these values will be noticed.

This provides a good platform for performing code execution trapping, though it is restricted to exactly four addresses. In addition, it may interfere with debuggers which wish to make use of the hardware support within the guest. A simple solution is to simply ignore access to the debug registers from the guest OS while such a mechanism is active. We argue that VMI is generally performed on production systems where extensive debugging is not required and since software debuggers are not hindered, this is a feasible solution. In cases where these restrictions cause issues, Section V-C4 provides an alternative method for code execution trapping.

D. Countering Threats with Hardware Mechanisms

The mechanisms we described so far are intended to be used as hardware anchors for VMI-based security applications. Table I summarizes which of these mechanisms can be used to counter which threat from Section IV.

VI. Conclusion

In this paper, we argued that derivative methods should be used when performing VMI whenever possible. A VMI application that roots its view-generation in a derivative anchor improves robustness against circumvention. Any time introspection is rooted in the (virtual) hardware, there is no way for an adversary to circumvent the introspection by moving to a lower level. As an adversary tries to circumvent introspection, he will eventually reach the hardware. At this point, his only option is to attack the introspecting

Threat		*Security anchor*	
IV-A	Interrupt hooks	V-A2	IDTR and system call MSRs
		V-B3	System interrupt trapping
		V-B4	User interrupt trapping
		V-C2	Page fault exception
		V-C3	General protection fault
IV-B1	Process hiding	V-B1	Context switch trapping
		V-A1	Control register 3 (CR3)
IV-B2	Module hiding	V-B4	User interrupt trapping
		V-C2	Page fault exception
		V-C3	General protection fault
		V-C5	Debug exception
IV-B3	Network activity hiding	V-A4	Virtual I/O
IV-B4	Virtual machine rootkits	V-B2	Virtual machine entry trapping
IV-C	Network-based attacks	V-A4	Virtual I/O
IV-D	Malicious processes	V-A4	Virtual I/O
		V-B4	User interrupt trapping
		V-C2	Page fault exception
		V-C3	General protection fault
		V-C4	Invalid opcode exception
		V-C5	Debug exception

Table I
LINKING THE THREATS TO THE SUITABLE HARDWARE MECHANISMS

component itself, which is isolated from the adversary due to the properties of VMI, leaving the adversary out of options.

Rooting the view-generation in hardware does not have to be a complex task as shown in Section III and the advantages have been made clear. We presented a toolbox of building blocks that may be used to anchor view-generation in hardware. In addition to these building blocks, we provided a clear discussion as to how and when each of these should be used and for which threats they may be applicable.

We urge the reader to consider the advantages of derivative methods and to apply the building blocks presented here when realizing VMI applications. Building a derivative based VMI application is well worth the effort when possible.

REFERENCES

[1] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[2] J. Pfoh, C. Schneider, and C. Eckert, "A formal model for virtual machine introspection," in *Proc. of the 2nd ACM Workshop on Virtual Machine Security*. New York, NY, USA: ACM, 2009.

[3] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. of the 8th Workshop on Hot Topics in Operating Systems*. Washington, DC, USA: IEEE, 2001, p. 133.

[4] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction," in *Proc. of the 14th ACM conf. on Computer and communications security*. New York, NY, USA: ACM, 2007, pp. 128–138.

[5] B. Hay and K. Nance, "Forensics examination of volatile system data using virtual introspection," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 3, pp. 74–82, 2008.

[6] L. Litty and D. Lie, "Manitou: a layer-below approach to fighting malware," in *Proc. of the 1st workshop on Architectural and system support for improving software dependability*. New York, NY, USA: ACM, 2006, pp. 6–11.

[7] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proc. of the 17th conf. on Security symp.* Berkeley, CA, USA: USENIX, 2008, pp. 243–258.

[8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Antfarm: tracking processes in a virtual machine environment," in *Proc. of the annual conf. on USENIX '06 Annual Technical Conf.* Berkeley, CA, USA: USENIX Association, 2006, pp. 1–1.

[9] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proc. the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2008, pp. 91–100.

[10] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*, A. Oram, Ed. O'Reilly, 2005.

[11] *Subverting Vista Kernel for Fun and Profit*. Blackhat USA, 2006, last access: 04/19/2010. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf

[12] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. of the 1996 IEEE Symp. on Security and Privacy*. Washington, DC, USA: IEEE, 1996, p. 120.

[13] A. P. Kosoresow and S. A. Hofmeyr, "Intrusion detection via system call traces," *IEEE Softw.*, vol. 14, no. 5, pp. 35–42, 1997.

[14] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2009, last access: 09/17/2009. [Online]. Available: http://www.intel.com/products/processor/manuals/index.htm

[15] *AMD64 Architecture Programmer's Manual*, 2009, last access: 09/17/2009. [Online]. Available: http://developer.amd.com/documentation/guides/Pages/default.aspx

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proc. of the Linux Symp.*, vol. 1, Ottawa, ON, Canada, Jun. 2007, pp. 225–230.

[17] A. Srivastava and J. Giffin, "Tamper-resistant, application-aware blocking of malicious network connections," in *Proc. of the 11th international symp. on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer, 2008, pp. 39–58.

[18] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proc. of the 15th ACM conf. on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 51–62.