Secure In-VM Monitoring Using Hardware Virtualization

Monirul Sharif
Georgia Institute of Technology
Atlanta, GA, USA
msharif@cc.gatech.edu

Wenke Lee
Georgia Institute of Technology
Atlanta, GA, USA
wenke@cc.gatech.edu

Weidong Cui
Microsoft Research
Redmond, WA, USA
wdcui@microsoft.com

Andrea Lanzi
Institute Eurecom, Sophia Antipolis, France, lanzi@eurecom.fr
Georgia Institute of Technology,Atlanta, GA, USA, andrea@cc.gatech.edu

ABSTRACT

Kernel-level attacks or rootkits can compromise the security of an operating system by executing with the privilege of the kernel. Current approaches use virtualization to gain higher privilege over these attacks, and isolate security tools from the untrusted guest VM by moving them out and placing them in a separate trusted VM. Although out-of-VM isolation can help ensure security, the added overhead of world-switches between the guest VMs for each invocation of the monitor makes this approach unsuitable for many applications, especially fine-grained monitoring. In this paper, we present Secure In-VM Monitoring (SIM), a general-purpose framework that enables security monitoring applications to be placed back in the untrusted guest VM for efficiency without sacrificing the security guarantees provided by running them outside of the VM. We utilize contemporary hardware memory protection and hardware virtualization features available in recent processors to create a hypervisor protected address space where a monitor can execute and access data in native speeds and to which execution is transferred in a controlled manner that does not require hypervisor involvement. We have developed a prototype into KVM utilizing Intel VT hardware virtualization technology. We have also developed two representative applications for the Windows OS that monitor system calls and process creations. Our microbenchmarks show at least 10 times performance improvement in invocation of a monitor inside SIM over a monitor residing in another trusted VM. With a systematic security analysis of SIM against a number of possible threats, we show that SIM provides at least the same security guarantees as what can be achieved by out-of-VM monitors.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

General Terms

Security

Keywords

Virtual Machines, Secure Monitoring, Kernel Integrity, Malware

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'09, November 9–13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-352-5/09/11 ...$10.00.

1 Introduction

Kernel-level attacks or malicious programs such as rootkits that compromise the kernel of an operating system are one of the most important concerns in systems security at present. These attacks can modify kernel-level code or sensitive data to hide various malicious activities, change OS behavior or essentially take complete control of the system. In addition, kernel-level security tools can be crippled and made ineffective by these attacks. A large body of research has adopted virtual machine monitor (VMM) technology in an effort to mitigate such attacks because the higher privileged hypervisor can enforce memory protections and preemptively intercept events throughout the operating system environment.

A major reason for adopting virtualization is to isolate security tools from an untrusted VM by moving them to a separate trusted secure VM, and then use introspection [7,17] to monitor and protect the untrusted system. Approaches that passively monitor various security properties have been proposed [10, 11, 13, 19]. However, passive monitoring can only detect remnants of an already successful attack. Active monitoring from outside of the untrusted VM, which has the advantage of detecting attacks earlier and preventing certain attacks from succeeding, was enabled by Lares [16]. This is achieved by placing secure hooks inside the guest VM that intercept various events and invoke the security tool residing in a separate secure VM. However, the large overhead for switching between the guest VM, the hypervisor, and the secure VM makes this approach suitable only for actively monitoring a few events that occur less frequently during system execution.

Many security approaches require the ability to monitor frequently executing events, such as host-based intrusion detection systems (IDSs) that intercept every system call throughout the system, LSM (Linux Security Module) [23] and SELinux that hook into a large number of kernel events to enforce specific security policies, or even instruction-level monitoring used by several offline analysis approaches [4]. Due to the overhead involved in out-of-VM monitoring, many such approaches either are not designed for production systems, or are not created for VM's. While keeping a monitor inside the VM can be efficient, the key challenge is to ensure at least the same level of security achieved by an out-of-VM approach.

In this paper, we present Secure In-VM Monitoring (SIM), a general-purpose framework based on hardware virtualization features that enables security monitors residing in the same VM it is protecting to have the same level of security as residing in a separate trusted or secured VM. A security monitor in our framework retains the efficiency close to being inside the same VM by not requiring any privilege transfers when switching to the monitor for an intercepted event, and being able to access the system address space at native speed. At the same time, isolation is achieved by putting the monitor code along with its private data in a separate

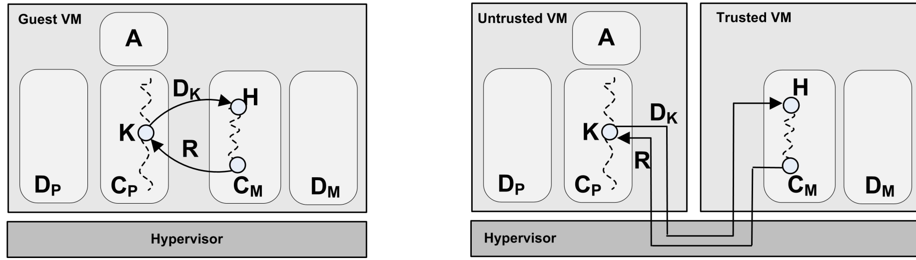

Figure 1: (a) In-VM and (b) Out-of-VM monitoring

hypervisor protected guest address space that can only be entered and exited through specially constructed protected gates. In other words, our system is designed in such a way that normal operation of the monitor can continue without hypervisor intervention, but any attempts to breach the security of SIM is trapped and prevented by the hypervisor. Our system design leverages Intel VT [9] hardware virtualization extensions and the virtual memory protections available in standard Intel processors.

We have developed a prototype of the SIM framework based on KVM [14], an open-source virtual machine monitor available as part of mainstream Linux that exclusively uses hardware virtualization features. Microbenchmarks show that an invocation of the monitor in SIM is almost 11 times faster than that of a monitor residing in a separate VM. As a demonstration of our framework, we have developed a process creation monitor that allows a selected list of processes to be executed. According to our microbenchmarks, out-of-VM monitoring introduced an overhead of 690%, which includes external introspection costs, whereas the overhead introduced by SIM was only 13%. We have also developed a standard system call interception monitor for the Windows OS. Macro benchmarks carried over a number of representative benchmark programs show an average overhead of 4.15% compared to 46.10% of out-of-VM system call monitoring.

We summarize the contributions of our work below:

- Leveraging hardware virtualization and memory protection features, we propose the Secure In-VM Monitoring framework where a security monitor code can reside inside a guest VM but still enjoy the same security benefits of out-of-VM monitors (see Section 2 and Section 3).

- We have implemented a prototype of the SIM framework based on KVM and Windows guest OS (see Section 4). For demonstration, we have developed two security monitoring applications using our framework. We provide experimental evaluation of the performance overhead. (see Section 5).

- We provide a systematic security analysis of SIM against a number of possible threats throughout the paper, and show that SIM provides no less security guarantees than what can be achieved by out-of-VM monitors.

The SIM framework should provide performance benefits to a large number of security monitoring tools that utilize virtual machines, and at the same time enable new applications that were not possible before due to the large overhead of out-of-VM monitoring.

2 Background and Requirements

A large fraction of security problems today are caused by kernel-level attacks or malicious code such as rootkits that violate some form of security in the entire system. Security tools such as anti-viruses, intrusion detection systems, and security reference monitors (e.g., SELinux) use various forms of event handling in a system in order to verify the system's security. We use the term *monitor* to represent the class of all security tools that either actively intercept events or passively analyze a system for violations of security.

If a monitor resides inside the same operating system it protects, the monitor itself can be compromised by kernel-level attacks. This problem is addressed by using virtualization. In virtual machine monitors, the hypervisor intervenes executions of the guest VM to give a virtual view of the real hardware. This is performed by utilizing the higher privilege of the hypervisor over the kernel to intercept accesses to the underlying hardware. Previous software-based virtualization utilizes a reduced privilege guest kernel, so that a higher privileged hypervisor can exist without any hardware support for virtualization. Recent processors include hardware virtualization [9] features to enable thin and light-weight virtual machine monitors. The privileged hypervisor allows various security approaches to protect access to hardware in an untrusted guest VM, and have security tools isolated from the untrusted system by placing them in a separate VM.

We illustrate two approaches of security monitoring in Figure 1. Figure 1(a) shows the model when the monitor resides in the same untrusted environment. We call this *In-VM* approach. The isolated monitor approach in a separate VM is shown in Figure 1(b), which we call *Out-of-VM* approach. At the high-level, In-VM approach provides performance and the Out-of-VM approach provides security. We define the performance requirements based on the In-VM approach and the security requirements from the Out-of-VM approach. The goal of our work is to design a system that satisfies both the performance and security requirements.

We present a few formal notations for precision and clarity in the following discussion. Assume that a system P is being monitored by a security monitor M. The system contains code C_P and data D_P and the monitor has its code C_M along with its private data D_M. For *passive monitoring* the monitoring code C_M usually needs to analyze the system code and data and use its own data to verify the security of the system. For *active monitoring*, since an event needs to be intercepted, a set of hooks $K = \{k_1, k_2, ..., k_n\}$ are placed in the monitored system that invokes corresponding *handlers* in the set $H = \{h_1, h_2, ..., h_n\}$ contained the monitoring code C_M. A hook can pass data D_K related to the event that is gathered at the point of the hook. After the handler handles the hooked event, a response R can transfer control to any specific point in the system. It is obvious that the active monitoring model subsumes the passive monitoring case.

The overhead in executing security tools out of the guest OS is primarily due to the change in privilege levels that occurs while switching back and forth between the kernel-level and the hypervisor-level. We set the performance requirements for SIM's design to be the same as provided by In-VM approaches.

- (P1) **Fast invocation:** Invoking the monitors handler H for a hook K should not involve any privilege level changes.

- (P2) **Data read/write at native speed:** The monitor code C_M should be able to read and write any system data D_P and

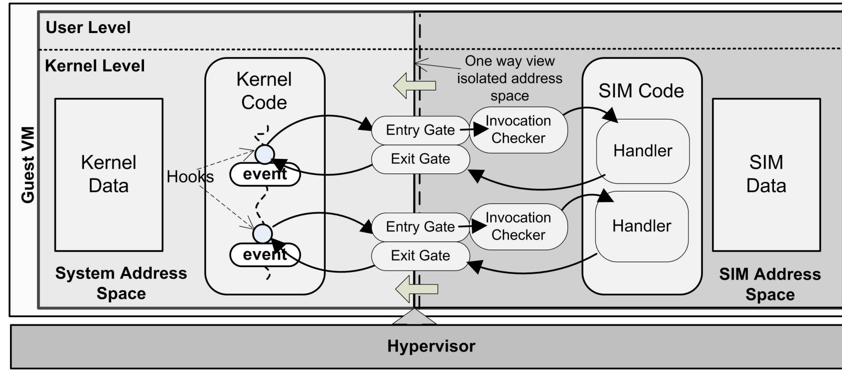

Figure 2: High-level overview of the Secure In-VM Monitoring approach

local data D_M at native speed, i.e., without any hypervisor intervention.

In case of in-VM monitoring, a direct control transfer to the handler code from the hook initiates the monitor. Moreover, the monitor can access all data and code because everything is contained in the same address space. The problem of out-of-VM approaches is that both performance requirements (P1) and (P2) cannot be satisfied. First, the hypervisor is invoked when the hook K is executed to transfer control to the handler residing in another VM. Second, the hypervisor usually needs to be invoked to partially map memory belonging to the untrusted VM into an address space in the trusted VM for the out-of-VM monitor.

To state the security requirements, we consider an adversarial program A residing in the same environment as the system P. In the threat model, A runs with the highest privilege in the guest VM and therefore can directly read from, write to and execute from any memory location that is not protected by the hypervisor. To ensure the security of the monitor M, we state the security requirements:

- (S1) **Isolation of the monitor's code C_M and data D_M:** This ensures the integrity of the monitor's code and data is protected from the adversary A. Out-of-VM approaches satisfy this requirement because A does not have any means to access another guest VM.

- (S2) **Designated point for switching into C_M:** Execution should switch to the monitor only at one of the handlers in the set H. This requirement ensures that an attacker does not invoke any code in C_M other than the designated points of entry. Since the hypervisor initiates entry into the monitor, out-of-VM approaches can ensure this requirement.

- (S3) **A handler h_i is called if and only if the corresponding hook k_i executes:** This requirement has two parts - (a) If a hook k_i is reached in the monitored system, then the corresponding handler h_i must be initiated by the system. (b) an handler h_i is initiated only if the hook k_i was executed. In out-of-VM approaches, the first requirement can be satisfied by design of the handler dispatcher. The second requirement can be satisfied because the exact `VMCalls` that initiated the hypervisor execution can be identified and checked.

- (S4) **The behavior of M is not maliciously alterable:** The execution of handlers H should not be maliciously alterable by the adversary A. First, the control-flow should not depend on any control-data that is alterable by the attacker. Second, the handlers should not need to call any dependencies that is at the control of the adversary. Third, after the handler completes, execution should return to a point that is intended by the monitor. An out-of-VM monitor can satisfy these requirements by not using any control-data contained in D_P.

None of the existing in-VM approaches can satisfy all of the security and performance requirements at the same time. First, the simple method of write-protecting the monitor's code C_M can only work for stateless monitors, which do not have any private data. A second approach may be to write protect the private data D_M using help from the hypervisor. This, however, will require the hypervisor to trap every write to verify the instruction. The performance requirement (P2) is thus not satisfied. Finally, in-lined monitoring approaches such as CFI [1] and WIT [2] can instrument each control-flow or memory write operation, usually at compile time, so that integrity checks can be enforced at run-time. A comprehensive coverage of all the required instructions needs to be performed to guarantee that the security requirements are satisfied. Such a modification of all kernel-level code is an overkill to achieve the performance requirements of a general-purpose monitoring framework that may be utilized for hooking different types of events occurring in an OS kernel. Our SIM approach is designed with all the performance and security requirements in mind.

3 Secure In-VM Monitoring

The goal of our Secure In-VM Monitoring framework is to enable security monitors that meet all the performance and security requirements discussed in Section 2. In this section, we describe the design of the SIM framework.

3.1 Overall Design

The overall design of SIM is shown in Figure 2. The key idea of SIM is to introduce a separate hypervisor-protected virtual address space in the guest VM, which we call the *SIM Virtual Address Space*. This protected address space is used to place the security monitor. It exists in parallel to the virtual address spaces being utilized by the operating system. The virtual memory is mapped in such a way that it has a *one-way* view of the guest VM's original virtual address space. This means that the security monitor can view the address space of the operating system, but no code executing in the operating system can view the security monitor's address space. A number of *entry gates* and *exit gates* are the only code that can transfer execution between the system address space and the security monitor's address space. Hooks are placed in the kernel before specific events that transfer control to corresponding gates. The entry gate has an invocation checker module that checks who invoked the entry gate. Finally, the security monitor's code (e.g., handlers for each hook) and data are all contained in the SIM address space. Next we describe how we construct the SIM address space using paging-based virtual memory and hardware virtualization features in detail.

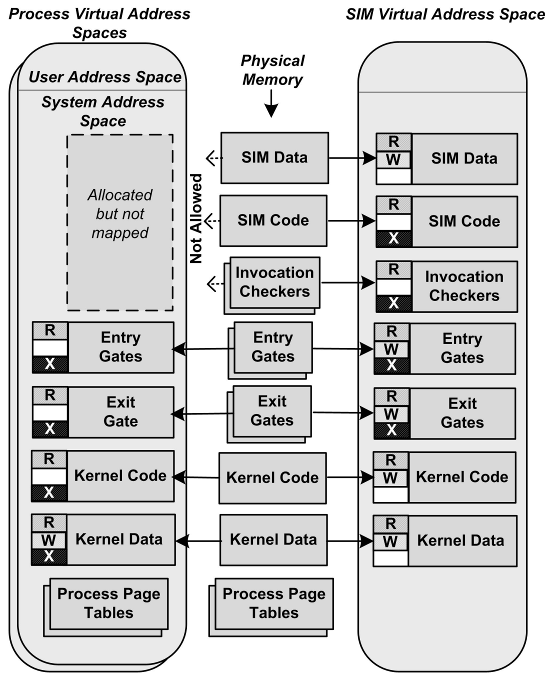

Figure 3: Virtual Memory Mapping of SIM approach.

3.1.1 Protected Address Space Generation

Paging based virtual memory is generated by creating page tables that map virtual addresses to physical addresses. When an instruction is executed the current page table is used by the hardware to perform address translations. An OS creates a separate page table for each process so that it can have its own virtual memory address space and the necessary isolation can be achieved.

The memory mapping introduced by the SIM framework is shown in Figure 3. In the figure, the *process virtual address space* at the left shows the virtual address space defined by the operating system for each executing process. The virtual address space created for the SIM is shown at the right as the *SIM virtual address space*. The actual physical memory regions are shown in the middle. For now, physical memory can be considered as guest physical memory. Later, we will describe how the address translations are carried out. We have only shown the relevant kernel level addressable regions, leaving user space out of the picture. For each region in the virtual address spaces, the protection flags that are set on the relevant pages by the hypervisor are shown.

A high-level description of the contents of the process virtual address spaces are shown in Figure 3. Generally, the kernel is mapped into a fixed address range in each process's address space. We call this address range the *system address space*. Since we are primarily focusing on kernel level monitoring, we illustrate the contents of this address range in the figure. We denote any code and data contained in the system address space as *kernel code* and *kernel data*. All pages containing kernel code will have read and execute privileges, but we assume that the kernel code can be write protected, especially places where hooks are placed. The data regions will have all access rights. In general, the dynamically loaded kernel-level code would be maintained in the kernel data region. In our framework, we introduce the entry and exit gates into the system address space. As mentioned earlier, the gates are used to perform transitions between the system address space and the SIM address space. Since the gates include code, they are set with execute permissions but are made read only so that they cannot be modified from with the guest VM.

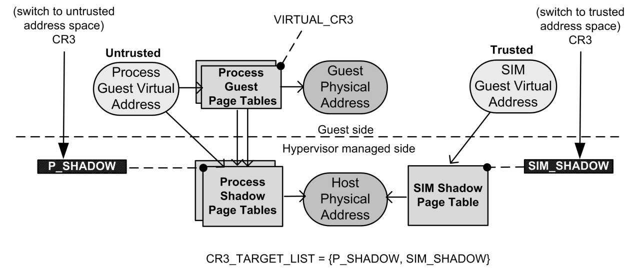

Figure 4: Switching between the untrusted and trusted address space without hypervisor intervention.

The SIM address space includes the security monitor's code (*SIM code*) and data (*SIM data*). Besides the security monitor, the SIM address space contains all the contents of the system address space that are mapped in. However, some of the permissions are set differently. The kernel code and data regions do not have execute permissions. This means that while execution is within the SIM address space, no code mapped in from the system address space will be executable. The invocation checking modules are also contained only in the SIM address space and have execution privileges.

Since the system address space contents are mapped into the SIM address space, an important requirement for the mapping to work is to ensure that other (the additional) regions in the SIM address space (i.e., the SIM code, SIM data and the invocation checker regions) do not overlap with the mapped in regions from the system address space. There are two methods to achieve it. First, the virtual address range that is used for user programs may be used for allocating the SIM regions. This approach is suitable for security monitors that will be primarily used to monitor kernel level code. Second, we can use OS provided functionality to allocate memory from the system address space. Once allocated, any legitimate code (including the OS itself) should not attempt to use this memory region in the system address space. Possible attacks may arise, which are discussed in Section 3.3.

Since the SIM address space contains all kernel code, data and also the SIM data in its address space, the instructions as part of the security monitor can access these regions in native speed. This satisfies the performance requirement (P2). The memory mapping method we have introduced satisfies the isolation security requirement (S1) by having the SIM code and data regions in a separate SIM address space. Any kernel-level instruction executing in the guest OS will utilize the system address space, which do not include these regions. Although any kernel-level code executing in the OS environment has full freedom to change the process virtual memory mappings because they are mapped into the system address area, they cannot modify or alter the SIM address space. By design, the SIM page table is neither included in the system address space, nor the SIM address space. In Section 3.1.2, we will explain the reason behind it and how we achieve it.

3.1.2 Switching Address Spaces

In the Intel x86 processors, the `CR3` register contains the physical address of the root of the current page table data structure In the two level paging mechanism supported in the IA-32 architecture, the root of the page table structure is called the *page directory* [8]. As part of the process context switching mechanism, the content of the `CR3` register is updated by the kernel to point to appropriate page table structures used by the current process. Although the kernel of the OS mainly maintains the valid CR3 values to switch among processes, any code executing with the kernel-level privilege can modify the CR3 register to point to a new page table. However, to ensure the correct operation of the operating system, kernel code needs to see its expected CR3 values.

In virtual machines, the page tables in the guest VM are not used for translating virtual addresses to physical addresses because the physical memory that needs to be translated to is on the host, which is maintained and shared among various VM's by the hypervisor. The hypervisor takes complete control over the guest OS memory management by intercepting all accesses to the CR3 register. The guest physical memory then only becomes an abstraction utilized by the hypervisor for maintaining correct mapping to the host physical address. *Shadow Page Tables* are used by the hypervisor to map guest virtual to host physical memory [3, 14, 21] and different mechanisms are used in different VMM implementations to maintain consistency among the guest page tables and the shadow page tables. The hypervisor gives the guest OS the illusion that its designated page tables are being used.

Since our mechanism requires the switching of address spaces, we need to modify the `CR3` register contents directly. However, the modifications to the `CR3` register by the guest VM is trapped by the hypervisor. The challenge is to bypass the hypervisor invocation, so that the performance requirement (P1) can be satisfied. For this reason, we utilize a hardware virtualization feature available in Intel VT. By default all accesses to the guest CR3 register by the guest VM causes a VMExit, which is a switch from the guest to the hypervisor. Intel VT contains a feature that it does not trigger a VMExit if the CR3 is switched to one of the page table structure's root addresses in a list (`CR3_TARGET_LIST`) maintained by the hypervisor [8]. The number of values this list can store varies from model to model, but the Core 2 Duo and Core 2 Quad processors support a maximum of 4 trusted CR3 values to be added in the `CR3_TARGET_LIST`.

The guest OS provides the addresses of guest page directories in the CR3 register, and the correct execution of the guest VM is ensured by the hypervisor changing them to the appropriate shadow page directories instead. However, if we bypass the hypervisor while switching CR3 values, we need to directly switch between the shadow page directories. Figure 4 illustrates how the switching is performed by updating CR3 register. Besides the hypervisor maintained shadow page table structures, we introduce an additional specialized shadow page table, which we call *SIM shadow page table*. The page table converts virtual addresses in the SIM address space to host physical addresses. Since it is directly maintained in the hypervisor and the security monitor need not manage its virtual memory, we eliminate the need for any guest level page table for the SIM address space. The root of the SIM shadow page table structure is the *SIM shadow page directory*, which we designate as `SIM_SHADOW`. We also designate the physical address of the current shadow page directory maintained by the hypervisor as `P_SHADOW`. Switching between the process address space and the SIM address space requires to directly modify the CR3 register and load the value of `SIM_SHADOW` or `P_SHADOW` after already adding them to the `CR3_TARGET_LIST`. This ensures the correct operation of the code in the guest VM when the hypervisor is not involved. The entry and exit gates described in Section 3.1.3 perform this switching, and the rest of the design of the SIM framework ensures that the switching is transparent to the guest OS.

3.1.3 Entry and Exit Gate Construction

The entry and exit gates are the only regions that are mapped into both the system address space and the SIM address space in pages having executable privilege. This ensures that a transfer between the address spaces can only happen through code contained in these pages. Moreover, since these pages are write-protected by the hypervisor, its contents cannot be modified by any in-guest code. The contents of the entry and exit gates are shown in Figure 5.

As mentioned earlier, each hook and handler have a pair of corre-

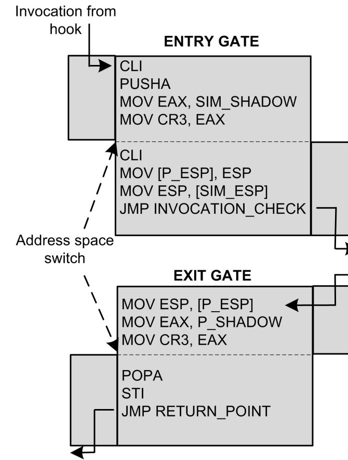

Figure 5: Entry and exit gates

sponding entry and exit gates. The task of an entry gate is to first set the CR3 register with the physical address of the SIM shadow page directory, or `SIM_SHADOW`. This switches the entry into the SIM address space. Since the CR3 register cannot be directly loaded with data, the value of `SIM_SHADOW` first needs to be moved to a general purpose register. For this reason, we need to save all register values on the stack, so that the security monitor can access register contents at the point when the hook was reached. Even though the register contents are saved on the stack on the system address space, since interrupts are disabled by the entry gate already, an attacker will not be able to regain execution and modify the values before entry into the SIM address space. Once in the SIM address space, the next task is to switch the stack to a region contained in the SIM data region by modifying the `ESP` register. The stack switching is necessary, so that code executing in the SIM address space does not use a stack provided by the untrusted guest kernel-level code. Otherwise, an attacker can select an address in the form of the stack pointer that may overwrite parts of the SIM data region once in the SIM address space. Finally, control is transferred to the invocation checker routine to verify where the entry gate was invoked (discussed in Section 3.1.4). Notice that the first instruction executed in the gate is the `CLI` to stop interrupts from executing. This guarantees that execution is not diverted to somewhere else due to interrupts. The reason for executing the same `CLI` instruction again after entering the SIM address space is discussed in Section 3.3.

The exit gate performs the transfer out of the SIM address space into the process address space. First the stack is switched back to the stack address saved during entry. To make the address space switch, the CR3 register is loaded with the address in `P_SHADOW`, which is the physical address of the shadow page table root. The hypervisor may be using multiple process shadow page tables and switching between them as necessary. To ensure correct system state, the value of `P_SHADOW` should be equal to the address of shadow page directory being used by the hypervisor prior to entering the SIM address space. Querying the hypervisor for the correct value during monitor invocation violates the performance requirement (P1). We take the approach of making the hypervisor update the value of `P_SHADOW` used in the exit gates when it switches from one process shadow page table to another. Having the value of `P_SHADOW` as an immediate operand in every exit gate would require the hypervisor to perform several memory updates. Instead, storing it as a variable in the SIM data region requires only one memory update by the hypervisor at the time of shadow page table switches. At the end of the exit gate, the interrupt flag is cleared to

enable interrupts again, and then execution is transferred to a designated point usually immediately after the hooked location. The exit gates have write permissions in the SIM address space, enabling the security monitor to control where the execution is transferred back.

The entry gates are the only way to enter the SIM address space, and they first transfer control to the corresponding invocation checking routine, which then calls a handler routine. By doing so, we ensure the security requirement (S2). Moreover, the "if" part of the requirement (S3a) is satisfied because, when a hook is executed, the corresponding handler is invoked. Additional variations of attacks are also handled by our design and are discussed in Section 3.3.

3.1.4 Checking Invocation Points

To satisfy the security requirement (S3b), once the SIM address space is entered through one of the entry gates, the invocation of the gate needs to be checked to ensure that it was from the only hook that is allowed to call the gate. The challenge is that since the entry gate is visible to the guest OS's system address space, a branch instruction can jump to this location from anywhere within the system address space. Moreover, we cannot rely on call instructions and checking the call stack because they are within the system address space and as such the information cannot be trusted. We utilize a hardware debugging feature available in the Intel processors after Pentium 4 to check the invocation points. This feature, which is called *last branch recording*(LBR) [8], stores the sources and targets of the most recently occurred branch instructions in some specific processor registers.

The last branch recording feature is activated by setting `LBR` flag in the `IA32_DEBUGCTL` MSR. Once set, the processor records a running trace of a fixed number of last branches executed in a circular queue. For each of the branches, the IP (instruction pointer) at the point of the branch instruction and its target address are stored as pairs. The number of these pairs stored in the LBR queue varies among the x86 processor families. However, all families of processors since Pentium 4 record information about a minimum of four last branches taken. These values can be read from the MSR registers `MSR_LASTBRANCH_`k`_FROM_IP` and the `MSR_LASTBRANCH_`k`_TO_IP` where k is a number from 0 to 3.

We check the branch that transferred execution to the entry gate using the LBR information. In the invocation checking routine, the second most recent branch is the one that was used to invoke the entry gate. We check that the source of the branch corresponds to the hook that is supposed to call the entry gate. Although the target of the branch instruction is also available, we do not need to verify it if the source matches. Our design also mitigates possible attacks that may jump into the middle of the entry gate and try to divert execution before invocation checking routine is initiated. This is discussed in Section 3.3.

A conceivable attack may be an attempt to modify these MSR registers in order to bypass the invocation checks. We need to stop malicious modifications to these MSR, but at the same time ensure that performance requirement is not violated. With Intel VT, read and write accesses to MSR registers can selectively cause VMExits by setting the MSR read bitmap and MSR write bitmap, respectively. Using this feature, we set the bitmasks in such a way that write attempts to the `IA32_DEBUGCTL` MSR and the LBR MSRs are intercepted by the hypervisor but read attempts are not. Since the invocation checking routine only needs to read the MSRs, performance is not affected.

3.2 Security Monitor Functionality

One of the important aspects of our design is to ensure that the security monitor code does not rely on any code from any untrusted region. Therefore, the security monitor code needs to be completely self-contained. This means that all necessary library routines need to be statically linked with the code and the monitor cannot call any kernel functions. From design, mapping the kernel code and data with non-execute privileges ensure that even any accidental execution of untrusted code does not occur in the trusted address space (because execution on non-execute code and data will result in software exceptions). Any software exceptions occurring while in the SIM address space is handled by code residing in SIM. Moreover, the entry and exit from the SIM address space can be considered an atomic execution from the perspective of the untrusted guest OS. While the hypervisor will receive and handle interrupts on the guest OS's behalf, they are not notified to the guest VM while the interrupts are disabled in the guest VM. Disabling interrupts before entering and after exit ensures that interrupts do not divert the intended execution path of the security monitor, which guarantees the security requirement (S4). Even without using the code of the guest OS, the same functionality provided by an out-of-VM approach can be achieved in our design.

First, by not allowing kernel functions to be called, the security monitor needs to traverse and parse the data structures in the kernel address space in order to extract necessary information required for enforcing or verifying security state of the untrusted region. However, this is the same semantic gap that exists while using introspection to analyze data structures of the untrusted guest VM from another trusted guest VM. The method of identifying and parsing data structures used in existing out-of-VM approaches can therefore be ported to our in-VM approach with a few modifications.

Second, a security monitor may need to perform accesses to hardware or perform I/O for usability purposes besides handling the events in the untrusted guest OS. Theoretically, it may be possible to replicate the relevant guest OS functionality inside the SIM address space. However, accessing hardware directly may interfere with the guest OS. We take a different step in our design. Since the SIM address space can be trusted, we allow a layer to be defined that communicates with the hypervisor for OS-like functionality through hypercalls. This layer, which we call the *SIM API*, can provide functionalities such as memory management, disk access, file access, additional I/O, etc. This layer can be developed as a library that can be statically or dynamically linked with the security monitoring code based on the implementation. The handling of the SIM API can be performed in the hypervisor or it may be performed by another trusted guest VM. Since the security monitor can be designed to use such functionality less often than handling events in the untrusted guest kernel (e.g., buffering data), the cost of hypervisor invocation can be kept low even for fine-grained monitoring.

3.3 Security Analysis

We first summarize how the our design has met all the security requirements stated in Section 2 without sacrificing any of the performance requirements. SIM satisfies the security requirement (S1) by using the hypervisor to not allow the monitor code and data to be mappable to any untrusted address space in the guest VM. The monitor remains in a completely isolated trusted address space isolated from the attacker. The requirement (S2) is satisfied because by design, the only method to enter the trusted address space from the untrusted one is via the entry gates. Since each hook invokes a corresponding entry gate, which eventually calls a corresponding handler, and each invoker of the entry gate is checked by the invocation checking routine, the requirement (S3) is satisfied. Finally, by not allowing any code from the untrusted domain to be executable in the trusted address space, and by design not allowing the monitor to call into the untrusted kernel, we ensure that the security

requirement (S4) is met. In the rest of this section, we discuss a few variations of attacks that are also handled by the design.

One important design consideration is to stop attacks that may divert the control-flow of the security monitor by modifying the control-data. Since any data in the untrusted region are completely at the hands of the adversary according to our threat model, it is important that, by design, the security monitor stores all control-data that it needs in the SIM data region. Ensuring this does not in any way reduce the functionality of the security monitor.

An attacker may directly call the entry gate and skip the first interrupt disabling instruction with an intention of keeping interrupts enabled after entering the SIM address space and before the invocation routine is executed, causing undefined behavior in case the interrupts are not handled properly. Having another `CLI` instruction at the beginning of the entry into the SIM solves this issue.

Since the entry gate is visible to the guest OS kernel-level code, the value of the SIM_SHADOW is revealed to the attacker. Although this value is known to the attacker, it cannot be used to switch into the SIM virtual address space from the attacker's kernel-level code. This is because, once the instruction that loads the CR3 register with the SIM shadow page directory's address is executed, the address space is switched immediately. The instruction immediately following the load instruction is from the new address space. Since the attackers code will not have execute privilege in the SIM address space, an exception will be generated. One possible modification of the attack is to allocate a page in the system address space that precedes immediately before a page that is used inside the SIM address space. If the instruction for setting the CR3 is the last instruction in the page, the next instruction executed will be a valid address inside the SIM address space. In order to defeat this attack, we ensure that each page whose immediate previous page does not contain code or have the executable privilege if it is allocated in the system address space. This ensures that such illegal entry into the SIM address space can be prevented.

4 Implementation

We have implemented a prototype of the SIM framework We used KVM (Kernel Virtual Machine) [14] for implementing the hypervisor component of the SIM framework on a Linux host with 32-bit Ubuntu distribution. The implementation was done on a system with Intel Core 2 Quad Q6600 processor, which has Intel VT support. We targeted Windows XP SP2 as the guest OS. To generate the SIM address space and load a security monitor into it, we rely on an *initialization phase*. Then, during the system runtime, the hypervisor based component provides memory protection, updates exit gates and handles VM calls that are relevant to the SIM API. We describe the initialization phase in Section 4.1 and the execution phase in Section 4.2.

4.1 Initialization Phase

The initialization phase of our system is initiated by a guest VM component implemented as a Windows driver that is executed after a clean boot when the guest OS can be considered to be in a trusted state. The primary task of the initialization driver is to allocate guest virtual memory address space for placing the entry and exit gates based on the hooks required, initiate creation of SIM virtual address space, initiate the loading of the security monitor into the address space, and finally the creation of entry gates, exit gates and invocation checking routines. The initialization driver communicates with the hypervisor counterpart of SIM using a hypercalls. We use the `VMCALL` instruction of Intel VT for the hypercall and use the four general purpose x86 registers to store arguments.

The first task is to reserve virtual address ranges in the system address space for use in entry and exit gate creation. Since we need to guarantee that the normal operation of the OS and legitimate applications do not attempt to utilize the reserved address ranges, we rely on the guest OS to allocate virtual address space. The driver allocates contiguous kernel-level memory from the non-paged pool by using the `MmAllocateContiguousMemory` kernel function. The function returns the virtual address pointing to the starting of this allocated memory region. Since the function allocates memory from the Windows non-paged pool, it is guaranteed by the OS to be never paged out. In other words, the pages are mapped to guest physical frames that are not used until they are freed. Since the memory is already allocated, any legitimate application will not try to utilize this address space. The allocated virtual address space region is informed to the hypervisor component using a predefined hypercall notifying the starting address and the size of the allocated region. During execution, our system checks for any malicious attempts to utilize this address space or changes in memory mapping.

The next step is the creation of the SIM virtual address space by the hypervisor component of the SIM framework. Once the hypervisor is informed about the memory allocation, the SIM shadow page table structure is created. In the 32-bit implementation of KVM, we noticed that KVM's own shadow page tables are implemented not as regular two-level 32-bit page tables, but as three-level 36-bit PAE (Physical Address Extensions) [8] page table structures. The Intel processors support this type of page table structure to handle more than 4GB of physical memory. To keep implementation elegant, one of our goals was not to make extensive modifications to KVM's MMU (Memory Management Unit) code, which mainly handles the shadow page table algorithms. Rather than utilizing the MMU code of KVM, we wrote our own code to create, maintain and update our SIM shadow page table. Since we need to switch between the same type of shadow page tables, we also implemented the SIM shadow page table as a PAE 36-bit page table structure. The usage of the PAE page table also enabled us to set NX-bits on pages, even though the 32-bit page tables used by the guest OS do not support this feature. During generation of the shadow page table, the system address space is traversed in the current process page table and mappings of all relevant entries are added to the SIM shadow page table with appropriate privileges.

In our current prototype implementation, the method we used for loading a security monitor application into the SIM address space is to load the application as part of the kernel driver in the system address space and then inform the hypervisor to place it into the SIM address space. The driver performs a hypercall with the starting address of the monitor code and its size. This hypercall enables the hypervisor component to allocate entries in the SIM shadow page table structure and map the virtual address range to newly allocated host physical memory that holds the monitor code as the SIM code region. We keep the same virtual address range so that no address relocation needs to be performed. The allocated host physical memory is intentionally not correlated with any guest physical memory, making it impossible for any guest page table to try to map these regions. In the same manner, the monitor data region is mapped into the SIM address space as SIM data. In our current prototype, we only support static data regions. Supporting dynamic data regions requires additional engineering effort and is orthogonal to the current design of our framework with a focus on performance and security.

The final task is to create the relevant routines to perform switching into the SIM address space. The security application requires hooking into the kernel for invoking its handlers. Any form of hooking can be utilized by the handler, and the method of hooking is orthogonal to our framework. For each hook and handler, a hypercall is performed by the driver to inform the hypervisor about the hook instruction, the handler's address and the address where

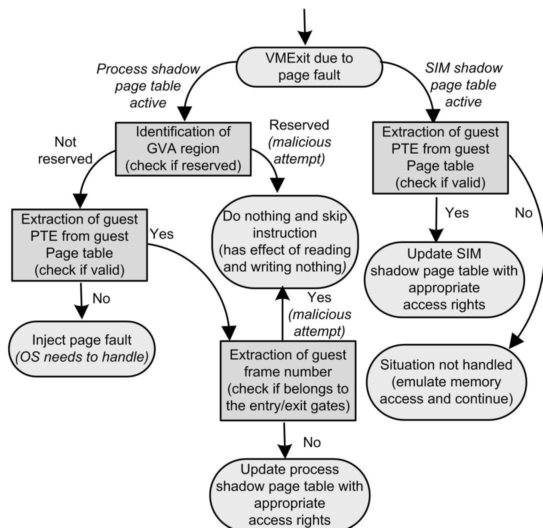

Figure 6: Run-time memory protection flowchart

to return execution to after the handler executes. For each received hypercall, the hypervisor component of our framework generates an entry gate, an invocation check routine, and an exit gate. The invocation checking routine is modified to verify the invocation instruction address to be the hook instruction address provided with the hypercall. A jump instruction is placed at the end invocation routine to jump to the provided handler. The exit gate code is also modified to return execution to the specified address. The address of the entry gate is returned, so that the driver can modify the hook to divert execution to the entry gate.

4.2 Run-time Memory Protection

At run time, attacks may attempt to breach the security enforced by the SIM framework by changing the virtual memory mapping in the system address space, or introducing a new virtual memory mapping. Rather than checking the guest page table entries for security violations whenever the guest CR3 is switched to a new page directory address, we ensure the security of the SIM framework by verifying memory protection whenever a guest page table entry is propagated to the shadow page table maintained by the hypervisor. Figure 6 illustrates the memory protection in our prototype.

Whenever a page fault occurs for the shadow page tables being used, a VMExit is called and trapped by KVM. The guest virtual address (GVA) for which the page fault occurred is identified by reading the guest CR2 register. We first check whether a process shadow page table or the SIM shadow page table is active. If the process shadow page table is active, we check whether the virtual address is in a reserved region (the address space reserved for SIM). If it is, it indicates a malicious memory access because this region is already allocated by the guest OS. We simply skip the instruction without doing anything, emulating the effect as if reading and writing succeeded. If the memory region is not reserved, we extract the guest page table entry (PTE) by traversing the current guest page table. If the PTE is not valid, we inject a page-fault into the guest, causing the OS to handle a regular page-fault situation. If it is valid, we then check whether the guest PTE is illegally trying to attempt to map the faulting GVA to guest physical memory containing entry or exit gates. In case of an illegal mapping attempt, we skip the instruction without doing anything. In case of a valid mapping, we update the process shadow page table. We ensure that the appropriate memory protection bits, such as write-protection, are enabled for PTE's updated in the shadow page table.

If the page-fault occurs while the SIM shadow page table is active, it must be due to accessing the kernel code, kernel data or user space regions in SIM since other SIM-specific code and data were already mapped in the initialization phase. Therefore, we traverse the current guest page table's system address space to search for the page table entry (PTE) corresponding to the faulting address. If the PTE entry is valid, in other words, the page is mapped to a valid guest frame number, we identify the host frame number and update the SIM shadow page table by inserting a corresponding PTE. However, we ensure that the mapped page does not have the execution privilege and has complete read-write access. If the PTE is invalid or is indicated as not present, it means that the guest OS possibly has paged-out the region. We currently cannot handle this case because it requires the untrusted guest OS to handle the page-fault. This is a violation of the security requirement (S4) because an attacker may change the behavior of guest page fault handler or insert its own. Our approach is to skip the instruction as if it succeeded without any effect. This problem can be overcome by having the hooking routine access these required memory regions before invoking the entry gate, causing all potentially paged-out frames to be paged-in before entering the SIM address space.

Table 1: Monitor Invocation Overhead Comparison

Monitor type	Avg. time (μ sec)	Std. dev. (μ sec)
SIM approach	0.469	0.051
Out-of-VM approach	5.055	0.132

5 Experimental Evaluation

5.1 Monitor Invocation Overhead

We performed micro benchmarks to compare the overhead in invoking an SIM monitor with an out-of-VM one. We measured the time required to switch to the monitor code from a hook and then switch back. For this reason, we implemented null event handlers that return immediately without performing any useful task.

For measuring our SIM framework, we implemented a security monitor whose handler only calls the corresponding exit gate. Hence, we could measure the monitoring overhead by invoking the entry gate, which caused a transition from the system address space to the SIM address space and back.

For measuring the out-of-VM overhead, we developed a simple inter-VM communication mechanism that enables a driver residing in the untrusted VM to invoke code of a driver placed in the trusted VM. Due to space constraints, we omit the implementation details. We measured the time taken from the hook being executed, initiating the communication from the driver in the untrusted VM to receiving execution back with a response from the trusted VM.

Table 1 shows the results of our micro benchmarks. Invoking the SIM monitor is almost 11 times faster than the out-of-VM monitor. We acknowledge that the overall performance gain will also depend on the processing time required for each invocation. A monitor that is frequently invoked with a small amount of processing in each invocation (i.e., fine-grained monitoring) should benefit from the SIM framework. In cases where the monitor is less frequently invoked and a large amount of processing is performed in each invocation, our SIM approach may have less performance advantage. However, this experiment only shows the switching overhead between the monitored system and the security monitor, leaving out additional performance penalties out-of-VM approaches may face due to external introspection costs. The experimental results in the Section 5.2.1 will show a more practical comparison of performance overhead by taking into account both switching and introspection costs for a real-world security monitoring application.

5.2 Security Application Case Studies

We developed two security applications using the SIM framework to perform a more elaborate and practical evaluation of the performance advantages of the SIM approach over an out-of-VM approach. We first developed a process creation monitor using our SIM framework and the out-of-VM approach. The monitor intercepts the creation of each process in the untrusted domain and allows execution if the name of the process is contained in a whitelist. Micro benchmarks on this application will not only take the monitor invocation overhead into account, but also consider any overhead introduced due to introspection performed by the out-of-VM monitor. Section 5.2.1 shows performance evaluation using this application. Although micro benchmark tests allow comparison of overhead introduced for monitoring individual events, it does not show the impact of monitoring on an overall system where several events are monitored continuously. Since system call events occur frequently, we developed a system call monitoring tool and performed macro benchmarks on a number of representative applications running in the monitored guest VM. We provide evaluation of SIM using this application in Section 5.2.2.

Table 2: Process creation monitor performance results

Monitor type	Average time (μ sec)	Relative overhead (%)
Traditional	3.487	×
Out-of-VM approach	28.039	690.5%
SIM approach	3.967	13.7%

Table 3: System call tracing macrobenchmarks

Benchmark	Bare Bare	Out-of-VM approach	SIM approach
Memory Latency	10.42 MAcc/s	84.58%	7.97%
HTML Render	1.12 pg/s	52.42%	5.83%
File Compress	3.4 MB/s	3.97%	0.59%
File Encrypt	20.56 MB/s	7.85%	0.89%
File Decrypt	78.21 MB/s	2.53%	0.45%
HDD	15.29 MB/s	41.68%	3.74%
Text Edit	82.73 pg/s	128.84%	9.64%
Average	×	46.10%	4.15%

5.2.1 Process Creation Monitor

The process execution monitor that we developed works similarly to the application presented in Lares [16]. We hooked the `NtCreateSection` system call in Windows by modifying the *System Service Descriptor Table* (SSDT). This system call is always invoked when a new process is created. The seventh argument to the system call is the handle related to the executable file, which lets the executable's name to be determined. Although the identification of the executable's name from the handle is straightforward using Windows kernel functions, we had to traverse the kernel data structures directly to extract this information in both the SIM-based and out-of-VM approach-based security monitors.

The security monitoring tool first extracts the `ETHREAD` structure of the current thread of the current process by traversing the *Thread Information Block* (TIB) pointed to by the `FS` segment selector. The `ETHREAD` structure is traversed to extract the `EPROCESS` structure related to the current executing process that performed the system call. We then traverse the object handle data structures to identify the object relevant to the object handle sent as an argument to the `NtCreateSection` system call. Once this object is identified as a file, the path name is extracted and compared with a list of allowed process images predefined in the tool. Depending on whether it is allowed or not, the original system call handler is called or skipped with a failure indication, respectively.

The SIM-based tool was written to directly access the system address space to traverse kernel data structures. However, the out-of-VM version requires introspection like functionality. Since KVM does not come with an introspection API, we implemented a limited form of introspection functionality in KVM ourselves. A driver in a guest VM calls a hypercall to map a page in its current system address space to the physical frame corresponding to a virtual address in the system address space of another VM. We were able to support this functionality in the hypervisor by simply updating the shadow page table corresponding to the guest VM's current guest page table with the requested mapping.

The performance micro benchmark results of the process execution monitor are provided in Table 2. We compared the performances of the SIM-based and the out-of-VM monitor with a *traditional* monitor that resides in the guest VM without any isolation. The out-of-VM version has almost 700% overhead over the traditional one. Our analysis revealed that the additional overhead beyond the invocation cost was due to additional hypercalls made for introspection purposes. With our SIM-based implementation, we only incurred an overhead of 13.7%, which is magnitudes lower than the out-of-VM approach. This overhead is primarily due to the cost of switching address spaces, which also invalidates the processor TLB cache, requiring additional memory accesses to page table. The results show that, besides reducing switching costs (as shown in Section 5.1), our SIM approach provides significant gains in performance in practical settings by providing the ability to access the monitored kernel data structure directly rather than the costly introspection methods used by out-of-VM approaches.

5.2.2 System Call Tracing

To evaluate the performance advantages of using SIM when frequent invocations are necessary, we developed a system call monitoring security tool and measured the run-time overhead on a system using a set of macrobenchmarks. To hook every system call occurring in the guest VM, we modified the `SYSENTER_EIP` MSR. In our prototype, we did not perform any specific system call inspection but read the stack to emulate the processing time a system call check would require.

For our SIM-based tool, we modified the MSR to point to code snippet that disables interrupts, jumps to the entry gate for the system call handler inside the monitor and upon return from the exit gate, jumps to the original system call handler. The system call handler code in the SIM extracts the userland stack address from register values saved on the stack by the entry gate. The handler stores the system call number together with a fixed number of arguments on the stack in a circular queue statically allocated in the SIM data region. After every 10,000 system calls, a hypercall is made to output a trace of the system calls via the SIM APIs.

In the out-of-VM approach, we pointed the `SYSENTER_EIP` MSR to a code snippet that performs a hypercall with the syscall number and the current userland stack address (that contains the arguments of the system call). The hypercall initiates communication with a driver in another VM that performs introspection on the userland stack to retrieve the arguments. The driver places the system call information in the circular queue which is sent to a kernel log after every 10,000 calls.

For performing the macrobenchmarks, we chose PCMark 05 [18] as the benchmarks suite. We did not include any graphics or audio tests because the drivers in KVM mostly emulate these hardware components. We first ran the benchmark in a guest VM without any system call monitoring tool installed. Then we performed the same tests with our SIM-based monitoring tool and the out-of-VM

tool. The results are shown in Table 3. Regardless of applications, the SIM approach had overhead a magnitude less than the out-of-VM approach. The overhead varied significantly among the applications for both the SIM and the out-of-VM versions, which was primarily due to the varying rate of system call invocations. The average overhead introduced by SIM was 4.15% compared to 46.10% of the out-of-VM version.

Since we did not parse the system call arguments based on their types, the amount of introspection required by the out-of-VM approach was fairly limited. Only one introspection VMCall was required per system call. In a full-fledged system call tracer that utilizes the argument type information, the overhead for introspection will have a significant impact on performance and our SIM approach should then show an even larger degree of performance advantage over out-of-VM approaches.

6 Related Work

Virtualization technology has played an important role in systems security. The security benefits gained by using virtualization has been first studied in [12,15]. Several approaches have then been proposed to use virtual machines for providing security in production systems [6, 7, 13, 16, 19]. In general, these approaches utilize the advantage of isolation from the guest VM, and monitor some properties in the guest system to provide security. While passive monitoring has been widely used in the past, Lares [16] recently proposed the method of actively monitoring events in a guest VM. Lares provides the framework of inserting hooks inside the guest OS that can invoke a security application residing in another VM when a particular event occurs. The design of Lares enables complex and large security tools such as Antivirus programs [20] or intrusion detection systems to run on the security VM. However, the cost in communicating an event notification from one VM to another via the hypervisor makes it inappropriate for use in fine-grained active monitoring. Our approach is similar to Lares in inserting hooks inside the kernel code to invoke a monitor and provides the same security benefits. However, in our approach the monitor invokation does not require the hypervisor to be involved and the monitoring code executes with kernel-level privileges in the same guest VM. Moreover, we have the entire kernel address space visible to the monitor's address space, whereas Lares-like approaches would incur additional costs for requesting the hypervisors to map in memory belonging to another VM.

Previously most approaches used paravirtualization or software-based virtualization. The recently introduced hardware virtualization features have gained attraction in the security research community. One recent approach that uses hardware virtualization technology is Ether [4]. The goal of Ether is to a provide a transparent malware analysis environment by hiding side-effects introduced by the dynamic analyzer that monitors the run-time events at the fine-grained level. By using various features in Intel VT and carefully introducing only privileged side-effects in the guest VM, Ether can intercept accesses to these side-effects and hide them from the malware. While transparency is very important for offline malware analysis environments to prevent malware from evading analysis, it is not so for production systems. As any other security system installed in a production system, the kernel-level instrumentation and entry and exit gates of our mechanism are visible to the malware. The threat model in this scenario is the neutralalization of the security mechanism instead of evasion. Our security requirements are, therefore, targeted towards this threat.

Our SIM framework can be considered as a method for enabling inlined-reference monitoring(IRM) for the kenel where the in-lined monitor is protected using hardware features. IRM has been widely adopted as a faster method of ensuring safety properties in programs by including the monitor inside the program it is monitoring. This is a far more efficient way than to have a reference monitor that resides in the kernel for user-space programs, or in the hypervisor for kernel-space programs. Traditional approaches of ensuring the integrity of the monitor itself for IRM techniques has been to ensure specific data-flow and control-flow safety properties throughout the program. For example, SFI [22] (Software Fault Isolation) is a method for having untrusted program share the same address space and provide isolation. This is achieved by rewriting specific store operations at compile time so that the address is masked in a way that it cannot write to an address region. SFI is well suited for application programs that can be modified during compile time. Achieving it for the kernel, which might have varying policies, is a hard problem. Control Flow Integrity (CFI) [1] instruments control-flow instructions with checks and their possible targets with labels at compile-time so that at run-time the checks enforce control-flow to be in the static CFG of the program. Since CFI covers all control-flow instructions, it also prevents circumvention of any of its checks. XFI [5] is an extensible fault isolation framework that provides fine-grained byte level memory access control. These features along with the protection of the XFI monitoring code is achieved by combining SFI and CFI. Finally, WIT [2] (Write Integrity Testing) provides protection from memory corruption attacks by verifying whether targets of write operations are valid by comparing with a statically and dynamically defined color table. Since write operations also encompass control-data, it provides integrity of monitoring code as well as protection from control-flow attacks without requiring CFI. Our approach of SIM provides the same isolation of the monitoring code in the kernel without having to guarantee properties such as SFI or CFI for all kernel level code.

7 Conclusion

In this paper, we have presented SIM, a general-purpose Secure In-VM Monitoring framework that by design provides the same security guarantees of out-of-VM monitoring and yet incurs similar low performance overhead of in-VM monitoring. We described the design of SIM and presented a comprehensive security analysis.

We have implemented a prototype SIM on KVM, and performed benchmarks on two representative security monitoring applications to compare the SIM approach with a typical out-of-VM one. Our microbenchmark results show that the SIM framework can reduce monitoring overhead by almost 11 times if only monitor invocation time is considered. The microbenchmarks on an introspection-heavy security application shows that SIM only introduces an overhead of 13.7% compared to 690.5% for an out-of-VM approach. In terms of the overall overhead on a system with a large number of event hooks and hence frequent invocations of the monitor, SIM keeps the overall overhead below 10% while an out-of-VM approach has overhead as high as 128%.

Acknowledgments

The authors would like to thank Artem Dinaburg for his help and support during the implementation and the anonymous reviewers for their constructive feedback. This material is based upon work supported in part by the National Science Foundation under grants no. 0716570 and 0831300, and the Department of Homeland Security under contract no. FA8750-08-2-0141. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Department of Homeland Security.

8 References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS)*, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating System Principles*, October 2003.

[4] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *In Proceedings of The 15th ACM Conference on Computer and Communications Security (CCS 2008), Alexandria, VA, October 2008*, 2008.

[5] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[6] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2003.

[7] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2003.

[8] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 3B: System Programming Guide, Part 1*, January 2006. Order Number: 253668-018.

[9] Intel Virtualization Technology. `http://www.intel.com/technology/virtualization`.

[10] X. Jiang, D. Xu, and X. Wang. Stealthy malware detection through vmm-based Şout-of-the-boxŤ semantic view reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[11] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[12] N. L. Kelem and R. J. Feiertag. A separation model for virtual machine monitors. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1991.

[13] K. Kourai and S. Chiba. Hyperspector: Virtual distributed monitoring environments for secure intrusion detection. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, 2005.

[14] Kernel based Virtual Machine. `http://www.linux-kvm.org/page/Main_Page`. Last accessed Apr. 20, 2009.

[15] S. E. Madnick and J. J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of theWorkshop on Virtual Computer Systems*, 1973.

[16] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[17] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, pages 385 – 397, December 2007.

[18] Futuremark PCMark 05. `http://www.futuremark.com/products/pcmark05/`. Last accessed Apr. 20, 2009.

[19] N. L. Petroni and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM conference on Computer and Communications Security*, 2007.

[20] P. Szor. *The Art of Computer Virus Research and Defense*. 2005.

[21] VMWare Virtualization Technology. `http://www.vmware.com/`.

[22] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Interprocedural control dependence. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1993.

[23] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. 2002.