Lares: An Architecture for Secure Active Monitoring Using Virtualization

Bryan D. Payne Martim Carbone Monirul Sharif Wenke Lee
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332–0765
{*bdpayne,mcarbone,msharif,wenke*}*@cc.gatech.edu*

Abstract

Host-based security tools such as anti-virus and intrusion detection systems are not adequately protected on today's computers. Malware is often designed to immediately disable any security tools upon installation, rendering them useless. While current research has focused on moving these vulnerable security tools into an isolated virtual machine, this approach cripples security tools by preventing them from doing active monitoring. This paper describes an architecture that takes a hybrid approach, giving security tools the ability to do active monitoring while still benefiting from the increased security of an isolated virtual machine. We discuss the architecture and a prototype implementation that can process hooks from a virtual machine running Windows XP on Xen. We conclude with a security analysis and show the performance of a single hook to be 28 μsecs in the best case.

1 Introduction

As malware has become increasingly sophisticated over the past several years, it is no longer unusual to see it disable critical security services on a victim's machine. Researchers responded to this threat by moving security services into different virtual machines (VMs) [9]. Techniques such as introspection make this possible by bridging the semantic gap between the protected VM and other VMs running on the same platform. In particular, introspection and related techniques have been used to build a wide range of security tools including intrusion detection systems, memory and disk integrity checkers, and system monitors [14, 18, 22]. All of these tools share one thing in common: each relies on *passive monitoring*. Passive monitoring is when the security tool monitors by external scanning or polling. As a result, it is unable to guarantee interposition on events before they happen.

This fundamental limitation of passive monitoring means that it is not a sufficient technique for implementing a full-featured anti-virus, intrusion detection, or intrusion prevention system. Previous efforts to implement these types of systems within a protected VM have resorted to implementing the systems with crippled functionality. What was missing in these systems was the ability to do *active monitoring*. Active monitoring is when the security tool places a hook inside the system being monitored. When execution reaches the hook, it will interrupt execution and pass control to the security tool. Active monitoring can also be done outside of the system being monitored (e.g., network and disk monitoring), however these monitors are restricted to the semantic level provided by the disk and network device abstractions. In this work, we focus on systems that perform active monitoring at a higher semantic level by placing hooks in arbitrary kernel locations within the system being monitored. This type of monitoring is required to support state-of-the-art security tools.

Active monitoring is a hard problem in the types of virtualized security architectures used in recent research. The problem is that active monitoring requires security-critical code inside untrusted VMs. Since a major reason for moving to a virtualized architecture is to remove security-critical code from the untrusted VMs, this feels like a step backwards. Properly protecting this code is sufficiently challenging that some researchers have attempted to avoid the problem altogether [13], resulting in systems that can only detect attacks, not prevent them. Furthermore, recent work has focused on providing strong protections for the entire kernel code [32], but these do not protect kernel data so they are insufficient for protecting entire applications. If this were the end of the story, then security in virtualized architectures would be limited to passive monitoring and the resulting security tools would remain crippled.

This paper addresses the problem of secure active monitoring in a virtualized architecture. We show how the monitoring mechanisms can be implemented and protected. We do not consider new malware detection or prevention techniques as these areas are orthogonal to the research presented here. Any system that uses active monitoring, in-

cluding any future advances in the field, can benefit from the added protections that our work provides. The primary research contribution of this work is an architecture to perform secure, active monitoring in a virtualized environment. We show design techniques that allow installation of protected hooks into an untrusted VM. These hooks will trap execution in the untrusted VM and transfer control to software in the protected VM. This architecture is generally applicable to any system that requires secure and active monitoring, and it builds on prior work that described techniques for passively monitoring memory and file system data [22].

Ensuring the security of this system is non-trivial. Our design is a departure from traditional secure systems work in that we place hooks inside the untrusted VM, without layering them directly on top of trusted code. By limiting the functionality of the code placed inside the untrusted VM and providing specialized protection mechanisms, we are able to ensure the security of this approach. The functionality removed from the untrusted VM is then implemented in a secure VM, so the overall functionality of the system is not reduced. The protection mechanisms can be deployed as needed so that the system only uses more costly mechanisms when required. Using these techniques, we are able to thwart attempts by malware to disable security applications that use our monitoring architecture. We provide a complete security analysis of our architecture in Section 6.

Our architecture is designed for use in production systems. To meet the goals of this environment, our architecture is designed to prevent an attacker from disabling or circumventing (i.e., bypassing) any security-critical component. The design also allows hooks to be placed at arbitrary kernel locations and provides a low performance overhead. These three requirements guided the design of our system architecture and are motivated by a set of formal requirements we formulated to ensure the monitoring system is secure. Section 2.2 introduces these formal requirements. We discuss our architecture and its requirements in Section 3.

After designing the architecture, we implemented a prototype to show the viability of our approach. Commercial security applications will utilize hundreds of hooks, but each hook is constructed in a similar fashion. Our prototype shows how the system would work by implementing one hook that is triggered for each new process creation event. This hook is commonly seen in security applications, such as anti-virus applications [35], and is representative of the types of hooks these applications use. The prototype is built on the Xen hypervisor using Windows XP in the untrusted VM and Fedora 7 in the secure VM. Our evaluation verifies the effectiveness of our memory protection techniques and measures the time required to process a single hook to be 28 μsecs in the best case. We provide implementation and evaluation details in Sections 4 and 5, respectively.

The rest of this paper is organized as follows. Section 2 provides the motivation and formal foundation for our work. Section 3 provides details on our architecture and the related design considerations. Section 4 describes our prototype implementation, including some implementation details that would be useful to anyone reproducing this work. Sections 5 and 6 evaluate the security and performance of our approach and discuss the security properties of our architecture. Section 7 describes the related work. We conclude with Section 8.

2 Secure Monitoring

In this section we first look at previous approaches to perform secure host-based monitoring. These fall into two main categories: passive virtualization-based approaches and malware analysis. We consider the benefits of and drawbacks to these approaches and compare them to active monitoring. Next we provide a set of formal security requirements for any active monitoring system. These requirements are used to motivate our architectural decisions in Section 3. The section ends with a discussion of the threat model and assumptions used in designing our secure active monitoring architecture.

2.1 Previous Approaches

Virtualization technology has made it possible to provide security services with better protection by isolating them into separate, protected VMs. Research on techniques like memory and disk introspection [9, 22] have shown how to leverage this isolation to securely monitor a system's state. Memory introspection works by having a security domain map the physical page frames of an untrusted domain into its own address space. It allows security applications to have complete visibility over another virtual machine's raw memory state. Using introspection, higher-level code and data structures can be semantically reconstructed to provide an abstract view of the system's state. Disk introspection works in a similar fashion, allowing security tools to access and infer the disk's contents in a protected fashion. While introspection has many applications, it is fundamentally limited because it can only perform passive monitoring. Therefore, introspection alone is not sufficient for applications that rely on active monitoring, such as anti-virus tools and host-based intrusion prevention systems.

Recent work on malware analysis [12,21,40] uses a form of VM-based active monitoring to intercept and analyze the run-time behavior of malware in a controlled environment. Although active monitoring is an integral goal of such systems, the requirements and usage scenarios of these systems and our system differ, making malware analysis approaches unsuitable for use on production systems.

Malware analysis systems are primarily designed to monitor a large and comprehensive set of activities inside the guest VM at a very low-level. This approach is feasi-

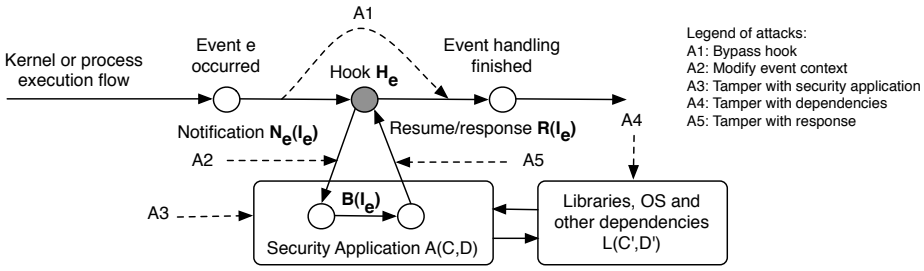

Figure 1: Formal model of secure active monitoring shown with potential attacks.

ble for two reasons. First, the offline fashion in which this analysis is normally done makes performance and run-time overhead a non-issue for such systems. Second, the fact that it is done in a staged, controlled environment, means that the collected low-level data can be directly mapped to the malware's activity with few false positives. In a production setting such as ours, the performance impact created by such systems make its use impractical. Furthermore, the low semantic level in which events are captured makes it difficult to infer the higher-level data needed to make security decisions.

In malware analysis, another important requirement is that the analyzer and the active monitor must remain hidden from the malware. This means that they should not introduce any noticeable side-effects in the malware's execution environment, as this could cause the malware to intentionally alter its behavior in an effort to thwart analysis attempts. However, in our production setting, we are only concerned with the protection and effectiveness of our monitoring components.

2.2 Formal Requirements

We present a formal model in this section that generalizes security applications performing active monitoring by placing hooks in a system to initiate actions when specific events occur. We use this model to analyze possible attacks on such applications under a powerful adversary that controls the entire system and identify a list of requirements that an ideal secure monitoring approach should satisfy in order to defeat such attacks. These formal requirements drive the design and architecture of our approach.

Figure 1 illustrates our model. Consider a security application $A(C, D)$ with code C and data D that wants to actively monitor the occurrences of a set of events E occurring inside a machine M. Suppose that the application depends on libraries or OS subsystems denoted by $L(C', D')$ for its execution. In our model, we generically represent events as activities occurring sometime along the execution of the kernel or a user process, which are handled by event-handlers that exist in the system. Any event $e \in E$ is actively intercepted by placing a hook H_e in the control flow path between the point of the event occurring and the point where handling the event is finished. The purpose of the hook is to initiate a diversion of control-flow to the security application. Depending on where the security application resides, this diversion can be a straightforward control transfer, a process switch or even an inter-domain communication. Therefore, we use a generic notation N_e to represent the notification call to the security application. The context information I_e about the event and the hook is sent along with the notification. We express the behavior of the security application for the particular instance of the event as $B(I_e)$, which may include performing checks, processing models, generating logs, determining appropriate responses, etc. Finally, the response of the security application is denoted by $R(I_e)$, which are actions carried out on the system, including updates to the state of the system or modifications in execution flow.

We can identify several classes of attacks on various aspects of the active monitoring model. The first class of attacks (A1) disables or bypasses the hooks H_e or tampers with the notification mechanism, so that N_e is not invoked. Attacks (A2) can target and modify the context information I_e, providing the security application with an altered view of the occurred event e. In addition, some attacks may change the behavior $B(I_e)$ exhibited by application on receiving I_e. These include attacks that modify the security application A and its code C and data D (A3), or any of its dependencies L (A4). Attacks (A5) may alter the response carried out by the security application by intercepting and modifying it.

The requirements for a secure active monitoring architecture that defeats the attacks are as follows:

1. N_e is triggered if and only if e occurs legitimately.
2. I_e is not modifiable between the occurrence of e and the invocation of N_e.
3. $B(I_e)$ of the security application is not maliciously alterable.
4. The effects of $R(I_e)$ on the system are enforced.

The first requirement states that an attacker should not succeed in circumventing hooks or generating spurious notifications. The second requirement ensures that an attacker cannot modify the context information I_e before invocation of N_e to alter or hide information regarding the event e. The third requirement ensures that the functionality of the security tool itself is not maliciously altered, defeating all attacks that tamper with the application process or any underlying subsystems it depends on. The fourth requirement ensures that the responses on the state of the system are always carried out as intended without letting the attacker modify them.

Assuming that an attacker has complete control over the system M, a security application that executes in the same machine with the same privileges as the attacker is unable to satisfy the above requirements. This is because the attacker can disable any protection mechanism and have complete access to the hooks, the application and its dependencies. By having higher privilege levels than the attacker, a hypervisor based approach can incorporate certain protections that the attacker cannot disable. Even in this scenario, if the security application is in the same VM as the attacker, it is hard to satisfy the third requirement. Since the application is running on subsystems controlled by the attacker, the run-time behavior of these subsystems along with the application needs to be protected, which may result in having to protect a large portion of the kernel. Although systems like SecVisor [32] may be sufficient to protect the kernel code, they are not suitable for protecting kernel data. This motivates our architecture of having the security application execute in a separate security VM, which is isolated from the attacker.

2.3 Threat Model and Assumptions

We make the standard assumptions seen in most other virtualization security architectures [6, 8, 9, 15]. The hypervisor and security VM are part of the trusted computing base (TCB) and the guest VM is not. Therefore, malicious code can only affect the guest VM. The hypervisor is ideally designed to be a small software layer that is both verifiable and secure. The hypervisor ensures isolation between the security VM and the guest VM, providing protection for security applications. Note that attacks such as Blue Pill [30] are not possible because a hypervisor using virtualization extensions is already installed as part of our architecture. Similarly, the SubVirt attack [17] is not possible because it was not designed to handle nested virtualization.

This paper is concerned with the runtime security of software placed in the guest VM. In order to focus on this problem, we also assume that the machine can undergo a secure boot [3]. Furthermore, we assume that the guest VM undergoes an *initialization* after boot. This initialization procedure will start the components, protect them, and provide for any additional security configuration. After the VM is initialized, it enters a *running* state where it is assumed to be subject to malicious software and other attack attempts. Our threat model is realistic and assumes that an attacker can do anything to the guest VM. This includes inserting malicious code into both application and kernel space.

3 Architecture

We call our architecture *Lares* after the Roman household gods that protected the home and family. Lares is a virtualization-based architecture designed to protect certain classes of security software that rely on the active monitoring of system events. Its design is based on the following architectural requirements:

1. **Protection of Monitoring Components**: The protection of the monitoring components should follow as closely as possible the formal requirements established in Section 2.2 for secure active monitoring.

2. **Flexibility in Hook Placement:** A security application built on top of Lares should have the flexibility to place hooks in any location in the guest OS's kernel, at arbitrarily high abstraction levels.

3. **Acceptable Performance Impact:** The performance overhead introduced by Lares should be within acceptable limits for the uses of most event-driven security applications, so as not to hurt overall usability and the performance of other applications.

The first architectural requirement for protection does not enforce complete adherence with the formal requirements raised in Section 2.2 due to the difficulty of preventing all possible attacks, as will be discussed in Section 6. It does, however, significantly raise the bar and prevents the majority of attacks against active monitoring security tools.

The overall structure of the architecture is illustrated in Figure 2, with components of the TCB represented in gray. Examples of applications that can benefit from Lares include anti-virus tools, anti-spyware tools and control flow-based intrusion detection systems. Other software that requires these types of hooks will also benefit.

3.1 Architecture Summary

There is a fundamental difficulty in conciliating protection and flexibility in hook placement. Flexibility allows hooks to be placed anywhere inside the untrusted system, which in a traditional scenario would make them prone to tampering by intruders with system-wide privileges. Solving this fundamental conflict is Lares' main contribution. At a high-level, it does this by splitting the security application into two VMs and using a special memory protection mechanism to guarantee the integrity of the hooks. As shown in

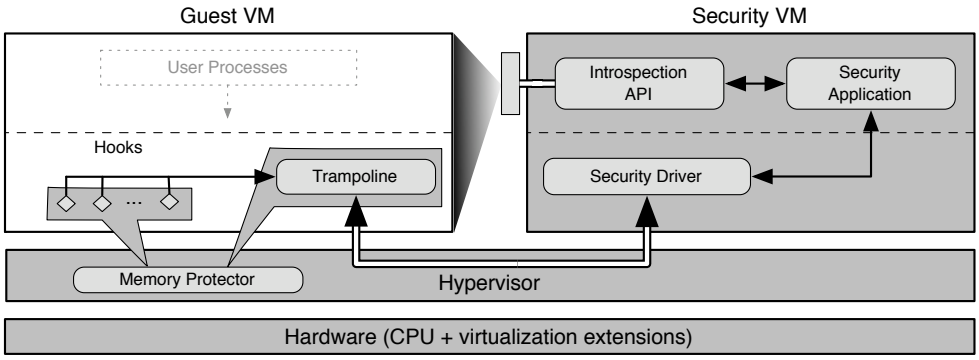

Figure 2: High-level view of the Lares architecture and its core components.

Figure 2, Lares includes two VMs: the untrusted guest VM and a security VM that is part of our TCB.

Since the guest VM is untrusted, software placed inside it requires special protection. This can be difficult to achieve if the components are too large or too integrated with the surrounding OS, so we keep them to the minimum required. These include the hooks for intercepting events, and a small specially-crafted trampoline code to pass events signaled by the hooks to the hypervisor. These components are self-contained and simple enough that write-protecting their memory footprint is sufficient to guarantee their correct behavior. We add a special mechanism to the hypervisor to provide these memory protections, along with an inter-VM communication functionality used for event passing. These additions, which we have implemented, are small to reduce the likelihood of introducing bugs into the hypervisor.

The security VM contains the core of the active monitoring application, where the processing and decision making associated with its functionality is done. Techniques like memory and disk introspection can be used as part of this decision making to gather additional information about events sent from the hooks in the guest VM. After a decision is made, the security application sends it back to the guest VM, where the decision is enforced.

As an example scenario, an anti-virus application would place its signature matching and containment algorithms in the security VM, whereas its monitoring hooks would go into the guest VM. These hooks would be triggered whenever certain monitored events were executed by the guest OS, and transmitted to the security VM by the trampoline with the aid of the hypervisor. The anti-virus' core engine would receive these events and use introspection to enrich them with contextual information, which would then be processed by its signature matching algorithms and heuristics. After reaching a decision, it would be sent back to the guest VM's trampoline, where a response measure is carried out, such as preventing a process from loading or a file from being written to disk.

3.2 Guest VM Components

In traditional systems, all applications are run within a single operating system. The guest VM fills the same role as this traditional operating system by running all applications that are not considered to be part of the TCB. The only exception is for the hooks and trampoline that are placed in the guest VM to achieve the active control and monitoring capabilities provided by Lares. Any application can be run in the guest VM since it runs a full featured operating system. With this in mind, the Lares architecture can be used to protect a wide variety of systems including servers and desktop systems.

One of the key capabilities in the Lares architecture is the ability to insert protected hooks throughout the operating system running in the guest VM. These hooks can be jumps placed inside program code, redirections within jump tables, or any other technique that transfers control of execution. Hooks are required for any security software that stops malicious code prior to it doing any damage. This is because other techniques can only monitor by polling and are unable to guarantee detection at arbitrary locations in the code. With the protected hooks, there is a guarantee that the security software can evaluate an action before allowing it to happen. This guarantee is provided by our memory protection mechanism, as described in Section 4.4.

When triggered, hooks redirect the system's control flow to another guest VM kernel component, the trampoline. The trampoline is a specially-crafted piece of code that acts as a bridge between the hooks and the security driver running in the security VM. It passes arguments from the hooked function to the hypervisor's inter-VM communication channel, which then delivers them to the security domain. The trampoline is also responsible for receiving commands from the security VM to execute actions requested by the security software. As the rest of the guest OS kernel is untrusted, the trampoline and the hooks must be protected from tampering. This need imposes several restrictions on the design and implementation of the trampoline. First, it must

be completely self-contained. This means that its functionality must not rely on any kernel functions or global variables, since these may be compromised. It must also execute atomically at each round, in order to prevent scenarios in which race conditions are used to circumvent the monitor. Finally, the trampoline's usage of data elements must be completely non-persistent (i.e., not rely on data that was generated in previous hooks activations). As our protection mechanism currently does not support the protection of data regions, not following this requirement would make the usage of such data prone to tampering.

3.3 Security VM Components

The security VM contains the back-end components of our architecture. These include the security application, the security driver, and an introspection API. The security application is where the decision-making functionality of the monitoring solution is implemented. It can be any software component that makes use of Lares, like an anti-virus tool or a host-based IDS. The security driver is the communications agent responsible for relaying events between the trampoline and the security application. These include hook notifications transmitted by the trampoline in the guest VM and relayed by the hypervisor, and decisions sent by the security application to the hypervisor. The introspection API provides the necessary introspection functionality to the security application, allowing it to collect additional information about the event that was trapped.

3.4 Hypervisor Components

Our architecture requires two special features from the hypervisor: protection of guest OS components and specialized inter-VM communications. The hypervisor modifications required to support these features are small, based on our implementation, which reduces the probability of introducing bugs into our TCB.

Guest OS Component Protection The protection mechanism is one of the key pieces of Lares, as it guarantees the integrity of the guest-space components of the architecture. Unlike other architecture components such as the security driver and security application, which are isolated by the architecture's inherent design, the hooks and the trampoline are situated in the guest OS's untrusted kernel. Therefore these components require special protection to prevent an intruder from tampering with their behavior. This type of tampering could involve the omission or forgery of events, or disabling the monitoring solution. Because the trampoline is self-contained and the hooks are jumps or function pointers, marking each hook's memory as read-only is sufficient to prevent tampering.

In a virtualized architecture, the hypervisor is an ideal place to implement such protections for two reasons. First, as we assume it is part of our TCB, it cannot be tampered by a malicious user. Second, as part of its job in virtualizing the hardware, the hypervisor has complete mediation power over the memory mappings used by the VMs running on top of it. Our architecture leverages this control to obtain a flexible, fine-grained memory protection mechanism. It is used to write-protect the hooks and the trampoline in the guest OS's memory, so that no tampering can occur with these components. A graphical representation of this protection is shown in Figure 2. The strength of this protection derives from the strength of the TCB itself: the only way an attacker could undo it would be to compromise the hypervisor, which we assume cannot be done.

Inter-VM Communication As our architecture requires components located in different VMs to communicate, inter-VM communication functionality is needed. As shown in Figure 2, the trampoline in the guest VM must send the events it captures from the hooks to the protector driver in the security VM, and the reverse path must also be traversed by replies sent from the protector driver. As virtualization inherently prevents VMs from directly interacting with each other, the implementation of such functionality must involve the hypervisor. The key property of the Lares communication mechanism that makes it different from existing generic mechanisms is that in Lares, the hypervisor must delay returning to the guest VM until a response is available from the security VM. A benefit of this design is that the guest OS will not be executing while we process the hook, which provides stronger guarantees for the system.

4 Implementation

Detecting malware on today's systems requires monitoring events as they happen. This, in turn, requires placing hooks throughout the system being monitored. These hooks are usually numerous and placed throughout the kernel to detect operations such as process creation, writing to disk, network activity, and inter-process communication. The Lares architecture is capable of placing hooks anywhere within the kernel of the guest OS. Hooking standard system calls requires the memory protections described in Section 4.4. Placing hooks in other locations requires additional protections as described in Section 6. Regardless of the hook location and its protections, the implementation of the hook processing system is the same.

Anti-virus monitors work by placing hooks in certain code sections and/or data structures inside the OS's kernel, so that key OS events can be trapped and analyzed. Trapping process creation events allows, for instance, to scan the image of the loading process for malicious signatures and prevent it from executing when a match is found. These hooks are placed for a wide variety of system calls and other critical events throughout the kernel [35]. For our prototype implementation, we choose a hook that is representative of the hooks used in these systems. Nearly all

anti-virus and host-based intrusion detection products place a hook in the kernel to monitor process creation. In Windows, NtCreateSection is the appropriate system call to hook for monitoring this event. The mechanism required to hook this system call is the same as hooking any other system call and very similar to hooking an arbitrary location within the kernel. Furthermore, the techniques used to process this hook are similar to what one would use for processing any hook. For these reasons, we chose to hook this location in our prototype implementation.

Our implementation uses Xen 3.0.4 for the hypervisor, Fedora 7 in the security VM, and Windows XP Service Pack 2 in the guest VM. We use an Intel processor with VT-x extensions. However, the architecture could also be built on an AMD system with the SVM extensions.

4.1 Hooks and Trampoline

In order to install the hook into the kernel API NtCreateSection, we implemented a Windows kernel driver called hookdriver.sys. Upon installation, which happens during the guest OS initialization, the driver creates a trampoline code section, modifies the appropriate system service descriptor table (SSDT) entry to point to it, and informs the hypervisor to activate necessary memory protections for the hook. The driver's implementation has 324 source lines of code (SLOC), and the trampoline occupies 89 bytes of memory.

The trampoline code is placed in a page of memory allocated from the nonpaged memory pool by calling the kernel function ExAllocatePoolWithTag. This ensures that the trampoline is always available, and will never be swapped to disk. The trampoline code section is copied from within the driver's code base to the newly allocated memory region. In order to modify the SSDT, we first identify the index of the NtCreateSection service. Then we identify the base of the SSDT using the kernel symbol KeServiceDescriptorTable. We then create the hook by placing the address of the trampoline in the appropriate entry after storing the old service routine's address (i.e., the location of the actual NtCreateSection function) in a pointer. This pointer is placed in the newly allocated memory region along with the trampoline code, so that it is protected from malicious modifications.

Once the hook is placed to point to the trampoline code, the driver initiates a notification call using a VMCALL to the hypervisor to inform the installation of the hook and the address range of the newly allocated memory region. This information is used to secure the indicated regions using the prot_range hypercall described in Section 4.4. This entire process is completed during the secure initialization of the guest OS.

4.2 Inter-VM Communication

When the trampoline code from the guest OS makes a VMCALL into Xen, it is sending a signal asking Xen to assist with inter-VM communication. In our architecture, inter-VM communication is facilitated by Xen with signaling from the domains performed through hypercalls. We added a new hypercall to Xen, lares_op, that is callable from both the guest VM (via a VMCALL instruction) and the security VM (via a direct hypercall). This hypercall takes two arguments. The first argument is a command. If the command requires a parameter, it is sent as the second argument. We provide details on each command below.

The LARESOP_security_register command saves a memory address of the buffer used to exchange information between Xen and the security driver. The other two commands are slightly more complex.

The LARESOP_guest_hook command builds a struct to send as a request to the security driver. This struct contains a unique identifier for the request and information about the hook event (e.g., hook number, associated Windows handle, or process id). This struct is copied to the security driver's shared memory region and then a virtual interrupt is sent to the security VM. This virtual interrupt, which is implemented using Xen event channels, is a signal to the security driver to process the hook information in its shared memory region. At this point, the command waits at a barrier until a reply is provided by the security driver. After the reply is provided, it is returned causing the VMCALL instruction to return, which allows the guest VM to continue normal operation.

The reply from the security driver is signaled with the LARESOP_security_response command. Upon receiving this command, Xen gets the reply value by copying a struct from the security driver's shared memory region. Next, the command makes this reply available to the hook command and breaks its barrier.

These three commands, implemented as a single hypercall, are all that is needed to support the inter-VM communication for the Lares architecture. They were implemented by adding 127 SLOC to Xen.

4.3 Security Driver and Application

The security driver and application are designed to execute within the same VM. For our implementation, this VM is the privileged para-virtualized VM called "domain 0" because it already has the necessary privileges to view memory from other VMs, thus simplifying memory introspection. However, it would be possible to implement this functionality in a fully-virtualized VM, or a different para-virtualized VM, if desired. Regardless of the location of these components, their function remains the same.

The security driver is designed to pass information up from Xen to the security application, and down from the

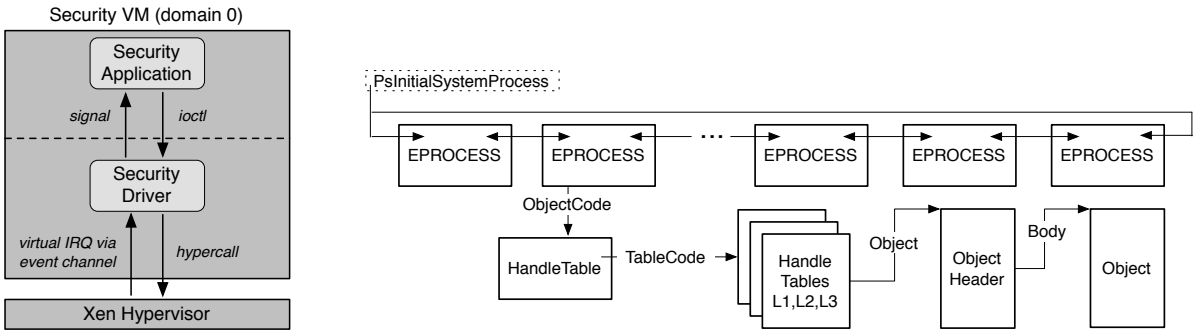

Figure 3: (Left) The information flow path from Xen, through the security driver, to the security application and back are all event driven to provide good performance when processing hook events from the guest VM. (Right) An overview of the data structures in the guest OS kernel traversed using introspection. The `TableCode` pointer is resolved at initialization time so that handles can be resolved efficiently for each hook.

security application to Xen as shown in Figure 3. No information processing or decision making occurs within this driver. This is an intentional design choice because it is harder to implement and change kernel-level code. As new features are added to this system, changes will typically only be made to the security application.

Since the security driver is designed to run in the Linux kernel, it is implemented as a Linux kernel module (LKM). The LKM is installed automatically when the security application is started. During initialization, the LKM sets up a shared memory region, a `proc` entry and its handler, and a virtual interrupt handler. The shared memory region is used to pass data between the security driver and Xen, as described in Section 4.2. The `proc` entry receives data from the security application and the virtual interrupt handler receives signals from Xen.

When a virtual interrupt is triggered by the `LARESOP_guest_hook` command, as described in Section 4.2, the security driver sends a signal to the security application. This signal tells the application that there is a new hook event to process. The LKM knows which process identifier (PID) to send the signal to because the application provides its PID as a module parameter to the LKM.

The security application is responsible for sending both the `LARESOP_security_register` command and the `LARESOP_security_response` command to Xen. After installing the LKM, the application sends a registration command to the driver by issuing an `ioctl` request through its `proc` interface. This command is forwarded to Xen using the `lares_op` hypercall. Likewise, after receiving a signal from the LKM, the application issues an `ioctl` request to receive the hook information and another `ioctl` request to send the response back to Xen.

After receiving the hook information, and before sending the response back to Xen, the security application must make a decision about how to handle this hook event. The decision is based on contextual information related to this hook event that is obtained using memory introspection. To access memory from the guest VM based on the guest OS's virtual addresses, we use the XenAccess library [22]. The hook that we implemented provides the security application with a Windows file handle that was passed to `NtCreateSection`. We lookup this handle in the kernel memory of the guest OS, and then extract the filename associated with the handle. Our implementation then allows execution of this file if it is contained within a white list of allowed executable files. More complex policies could also be implemented such as inspecting the file for a virus signature or to validate its checksum. The sophistication of these policies is only limited by the information available in memory, or on disk, in the guest VM.

Looking up the Windows file handle using introspection requires bridging a semantic gap. The memory structures we traverse are shown in Figure 3. We first identify the `PsInitialSystemProcess` which is a Windows `EPROCESS struct` representing the system process. Next, we traverse `ActiveProcessLinks`, which is a circular doubly linked list of all processes on the system. We identify the process associated with our hook, and then follow a series of pointers to the level 1 (L1) handle table. From this point, we walk the handle tables that are structured similarly to multi-level page tables. Next, we get the address of the handle's object header. The `ObjectHeader struct` contains a pointer to the object type. If the object is a file object, then we resolve the object, which contains the full path and file name. Finally, this file name is used to verify if the hook should be allowed.

The security driver and application are both written in C. The driver is 182 SLOC and should not require any changes when new hooks are added to the system. The application is 298 SLOC for our proof of concept implementation. More complex policies and the handling of additional hooks would be added to this code base, making such changes relatively simple to implement and debug.

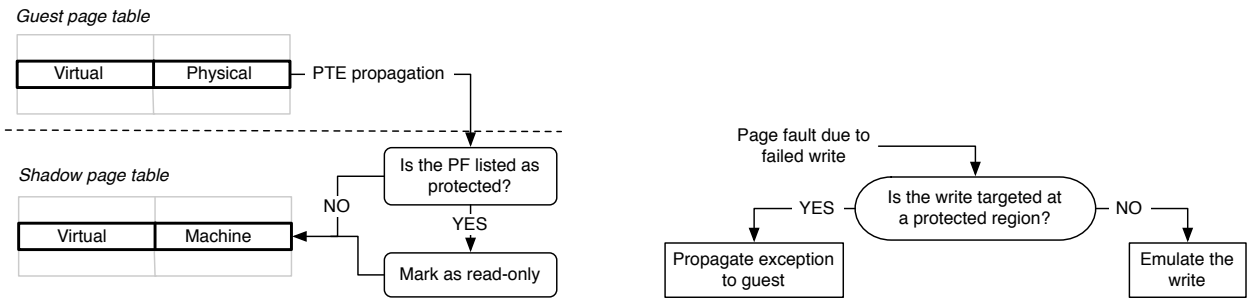

Figure 4: The page protection scheme leverages the propagation of page table entries (PTEs) from guest space to hypervisor space (left) to protect memory pages, as well as Xen's page fault handler to make sure it can be done with a byte-sized granularity by emulating the memory write (right).

4.4 Memory Protection

We leveraged Xen's memory management subsystem when building the memory protection mechanism. The primary goal of memory management in Xen is to virtualize each guest OS's view of memory and enforce isolation between the OSes. In fully-virtualized VMs, Xen does this using a technique called *shadow paging*. This technique maintains two versions of page tables for each VM: guest page tables (GPTs), which are controlled by the guest; and shadow page tables (SPTs), which are controlled by the hypervisor. The guest OS handles its GPTs the same as it would in a non-virtualized setting. The main difference is that the GPT's mappings translate virtual addresses to an intermediate layer of addresses, called *physical addresses*. Physical addresses virtualize the memory view of a guest OS, similar to the way virtual addresses work for processes. SPTs provide direct mappings from virtual to machine addresses, which are the addresses used by the hardware. Therefore, the SPTs are used by the hardware to translate addresses for the guest OS while Xen maintains consistency between the GPT and SPT. When an entry is added or changed in a GPT, Xen translates the physical address into its corresponding machine address, performs any necessary adjustments, and then updates the corresponding SPT. This process is called page table entry (PTE) propagation. Under this model, Xen controls the actual machine frames used by each VM, while also providing each guest OS with the illusion that it has full control of the memory.

Our memory protection mechanism protects arbitrary memory regions with a byte-sized granularity. It is composed of two main parts. The first is implemented in the `_sh_progatate` function, which controls the propagation of entries from GPTs to SPTs. The second is implemented in the `sh_page_fault` function, Xen's page fault handler. Our mechanism adds 78 SLOC to Xen, satisfying our requirements of making only minimal additions to it.

The first part, illustrated on the left of Figure 4, implements the core technique behind our protection mechanism. At this location we intercept the propagation of entries between the GPTs and SPTs, and then write-protect designated frames of the guest OS's physical memory. This can happen whenever an entry is modified in a GPT, either legitimately or by an attacker. Since Xen has full mediation over propagation and is isolated from the the guest OS, such protection cannot be circumvented. We store a list of the memory regions that require protection, which we call the protection list. Each time an entry is propagated from a GPT to an SPT, this list is searched for the entry's physical frame. If it is found, its corresponding shadow copy is marked as read-only. This prevents the guest OS from performing any further modifications to this page frame.

By itself, this technique only provides page-level protection, which is problematic if a page contains protected and writable regions. The second part of our mechanism extends this technique to provide byte-level protection. Its operation is illustrated on the right of Figure 4. Each time a page fault occurs due to a failed write, we check the target's virtual address, which is stored in the `cr2` CPU register. Next, we check the protection list to see if the target address requires protection. If it does, a page fault exception is propagated to the guest OS, preventing the write attempt. If not, then the guest is attempting to write to a non-protected region of a frame that contains a protected region. In this case, we emulate the write operation for the guest OS.

We added a new hypercall, `prot_range`, that can be called from the security application running in domain 0 to initialize the protection list. This is done from domain 0 during the architecture's initialization, as soon as the hook and the trampoline are placed inside the guest OS. Additional memory ranges can also be added to the list at runtime, if desired. Each time an update is made to the list, the shadow page cache is erased to eliminate outdated mappings and force the re-propagation of new mappings. Since most applications, including our prototype, only add items to this list during initialization, there is no runtime performance impact on the system.

This combination of page-sized memory protection and write emulation allows us to efficiently implement the protection of arbitrary memory regions of the guest OS, with the granularity of a single byte. In our prototype, we used this mechanism to protect several memory regions in the guest OS. The first was the `NtCreateSection` hook placed in the SSDT, a 4-byte long function pointer. The second was the trampoline, a segment of code consisting of 89 bytes in a memory page allocated when the architecture is initialized. Additional components that require protection to prevent hook circumvention are discussed in Section 6.

5 Evaluation

We tested both the security and performance of our prototype implementation. Security was tested by verifying that the memory protection techniques worked as expected. In addition, we provide an extensive analysis of our architecture's security in Section 6. Performance was tested by measuring the time required to process a hook using our architecture, compared with an architecture similar to that of current security applications.

5.1 Security

An essential component of a successful attack is to evade detection by either hiding itself or by disabling defensive measures altogether. Many malware today incorporate features, similar to Agobot, to disable commercial anti-virus programs. Earlier methods involved process termination system calls for known anti-virus process names. However, most anti-virus programs incorporate hooks into process termination and creation routines to monitor their usages patterns for self-defense [35]. In addition, hooks are placed into events such as file/disk access or Windows registry updates for detecting malicious updates to the system. To defeat such systems, malware programs incorporate various rootkit methods [10] that can remove such hooks to successfully disable anti-virus tools prior to infection. A recent work [39] allows automatic analysis of the hooking behavior of malware by identifying the modified code and data structures. A large number of malware place their own hooks in order to hide their processes, drivers, files (e.g. the FU rootkit, NT rootkit etc.) and their malicious changes in the system or backdoors (e.g. Uay Backdoor). A classification of rootkits can be found in [29].

As described above, a technique used by attackers to defeat commercial anti-virus tools is to replace the hooked function pointer in the SSDT to either the original function pointer (disabling the hook) or to a malicious function pointer that in turn calls the original one (hijacking the hook). In order to perform this attack, the malware must have detailed knowledge of the internal workings of these tools. Since these kinds of low-level attacks must be specially crafted for the target security tool to be effective, we were not able to test our memory protections with pre-existing attack code. Instead, we developed a synthetic attack that performs the hook hijacking attack using the same technique used by rootkits.

Our test system consisted of the guest VM running with the trampoline and hooks initialized. To ensure the synthetic attack works properly, we then ran it without any memory protections enabled. During this test, the synthetic attack worked as expected, hijacking the hook and effectively preventing any execution of the trampoline code. Next, we repeated the test with the memory protections enabled. This time the synthetic attack failed to complete its installation because it was unable to change the write-protected entry in the SSDT and the trampoline code continued to execute normally.

This test shows that the memory protections work properly. Similar attacks could be constructed to modify the trampoline code, the pointer to the SSDT in the system service dispatcher, or the pointer to the system service dispatcher from the IDT. However, these attacks would also fail because these regions of memory are also protected by our system. An analysis of additional security considerations is provided in Section 6.

5.2 Performance

Our benchmarking tests were run on a Lenovo Thinkpad T60p laptop. The test machine has a Intel Core Duo T2700 processor running two cores at 2.33GHz, 2KB of cache per core, and 2GB of system memory. The hypervisor was Xen 3.0.4. The guest VM ran Windows XP SP2 and allocated 384 MB of memory. The security VM ran Fedora 7 and allocated the remaining system memory.

Hook processing is the key operation where the Lares architecture will differ in performance from a traditional architecture. Therefore, our benchmark measurements look at the time required to process a single hook in the Lares architecture and compare that with a traditional architecture. To measure the hook processing time with the Lares architecture, we instrumented the trampoline code. We retrieved the value of the processor's performance counter before and after the `VMCALL` instruction. The processor's performance counter was obtained using a function provided by Windows, `KeQueryPerformanceCounter`. The difference between these two measurements represents the time needed for inter-VM communication and hook processing within the security VM. However, this measurement is noisy. It can be influenced by cache effects, VM scheduling, physical interrupts, CPU frequency scaling, and other loads on the system. We took several steps to minimize the influence of this noise in our measurements. First, we pinned each VM to its own CPU core. Next, we disabled unnecessary services in the security VM. Then we disabled CPU frequency scaling in the BIOS.

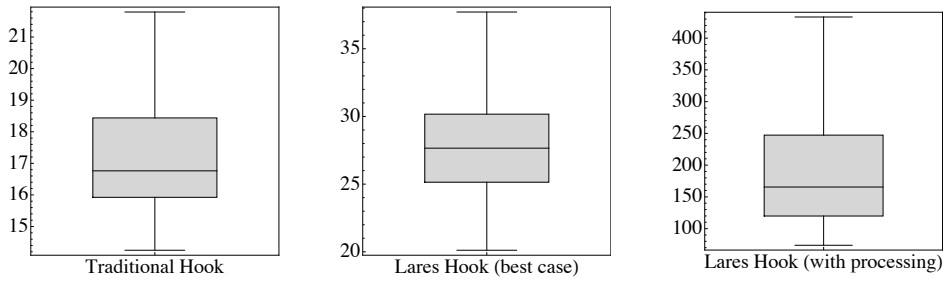

Figure 5: Hook performance is shown in the left three charts. The traditional hook shows the time for processing a hook locally. The Lares (best case) shows the time for just sending a notification to the security VM. The Lares (with processing) shows the time for processing the hook in our prototype application, using introspection to lookup file names from handles. All of these times are in μsecs.

After the system was prepared as indicated above, we measured the hook performance across five runs, where each run included 1000 measurements. The 5000 measurements were combined into a single data set. Then we performed a standard statistical analysis to remove the outliers, since there was still some noise in the measurement. Our analysis computed the inner-quartile range (IQR) of the data set and defined outliers to be 1.5 times the IQR above the third quartile and below the first quartile. After removing the outliers, the computed mean on our data was 28 μsecs to just send a notification to the security VM and 175 μsecs when the security application used introspection to lookup file names from handles. A more detailed look at the performance characteristics of memory introspection is available in our prior work [22].

To measure the hook processing time for a traditional architecture we developed a system that processes the hook inside the guest VM. The kernel code in this system is the same as the Lares architecture except that instead of executing the `VMCALL` instruction to process a hook, we send an event to a user-space application. This application performs the same check as our prototype, looking up the file handle to check the associated file name. The system was prepared and the tests were run the same as for the Lares architecture test. After removing the outliers, the computed mean on our data was 17 μsecs. The results from each of these tests are shown graphically in Figure 5. These graphs, known as boxplots, show the IQR as a gray box. The median value is denoted with a horizontal line through the box. And the range of the remaining nonoutlier data is shown as lines extending above and below the box.

Two major factors contribute to the differences in the performance of Lares versus a traditional architecture. First is the fact that it takes more time to exit the guest VM, send a signal to the security VM, and perform the software address translations needed for memory introspection than it does to perform the same tasks locally. This factor contributes to the overhead for a single hook event. The second factor is more subtle. When everything is processed locally as in the traditional architecture, there is no security benefit to processing data inside the kernel versus in application space. Since only a subset of the calls to `NtCreateSection` are associated with the file handle of a new process execution attempt, the traditional architecture can use the result of the `ObReferenceObjectByHandle` function in the windows kernel to filter out the hook events that do not need additional processing. Using this technique, only a subset of the hook events are sent to user space for processing. However, in the Lares architecture, we do not trust any local functions in the Windows kernel. So *every* hook event is sent to the security VM for processing. This example is specific to the hook that we implemented, but a similar situation would exist for other hooks as well. This trade-off raises the mean time required for hook processing in the Lares architecture, but also increases the security of our architecture by reducing dependencies on untrusted code.

While our benchmarks show that Lares is slower than traditional hook processing, we also provide a significant improvement in security. The overall performance of a given application will ultimately depend on the number of hooks it uses in addition to its use of introspection and other techniques to collect data for processing each hook. Our experience with the example application and the performance results presented in this section suggest that applications using Lares can perform similarly to applications using traditional architectures.

6 Security Analysis

In this section, we analyze the security of our prototype and show that the majority of the attacks discussed in Section 2.2 against secure active monitoring applications will not succeed against our architecture. We discuss each of these attacks using our prototype as the case study.

In our architecture, the security application, security driver and underlying OS in the security VM are inherently protected from any type of disabling or tampering (attacks A3 and A4) initiated from the intruder in the guest VM. This can be generalized to our architecture as a whole, as

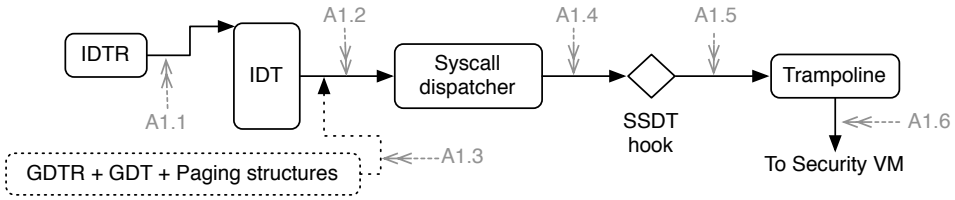

Figure 6: Various forms of A1 attacks aimed at circumventing our monitoring mechanism.

any application that makes use of Lares will be in the security VM, and therefore integrated with the TCB. As such, the only option for an attacker would be to target the guest-space infrastructure (attacks A1, A2 and A5) on which the security application relies on for active monitoring

We first discuss various forms of A1 attacks illustrated in Figure 6, which are aimed at circumventing our monitoring infrastructure. They work by maliciously modifying the guest architecture components and the system structures that they depend on. In case of the SSDT hook and the trampoline, disabling or tampering with such components (attacks A1.5 and A1.6) would mean maliciously altering their memory state. For example, the hook could be replaced by another that points to a malicious function, or the trampoline could have critical parts of its code (such as the VMCALL) erased. Attacks of this type against the hook and the trampoline are effectively blocked by our architecture, as we write-protect their memory region. This fact has been empirically verified in the experiments we conducted, presented in Section 5. We can also generalize this property to any type and number of hooks in the kernel, as our hypervisor-based memory protection is done at the byte level and can be applied anywhere in kernel code or data.

Other types of circumvention attacks illustrated in Figure 6 involve manipulating certain kernel structures which control the kernel's flow of execution between the moment at which the `NtCreateSection` event happens and the hook is triggered. In the case of our SSDT based hook, the IDTR register (A1.1), the system's IDT (A1.2), the system call dispatcher (A1.4) and current address translation structures (A1.3) can be targeted. The latter includes the GDTR register, the GDT and the current page table. These registers and structures are defined by the x86 architecture. The first three determine the flow of execution immediately after a system call is triggered by the `int` CPU instruction. In our prototype, we write-protect the IDT and the system call dispatcher code in memory, nullifying attacks A1.2 and A1.4. Such protections have no negative side-effects, as these structures are normally not supposed to be modified at run-time. For the IDTR, we were unable to implement a similar kind of protection, as changes to this register cannot be trapped by a hypervisor using Intel VT-x. Instead, we check its contents for modifications at every exit from the guest to the hypervisor. These exits (called VM exits) happens at least at every context switch, making attacks targeted at this register extremely difficult to succeed due to the short window of opportunity. However, for the AMD SVM platform, this attack is not a concern because changes to the IDTR are trapped by the hypervisor.

A more sophisticated type of circumvention can be done by manipulating the GDTR, the GDT and the system's page tables, as these are used in the translation step between the IDT and the system call dispatcher (attack A1.3 in Figure6). An attacker could tamper with such structures to manipulate the address translation, and redirect the flow of execution to a different physical address, containing malicious code. Memory introspection is affected by a similar issue, as it normally uses page tables inside the guest OS to perform memory translation. Although possible, such an attack would nevertheless be considerably difficult to implement in this particular case, as the dispatcher shares a single 4MB page with the rest of the Windows XP kernel. To succeed, an attacker would thus need to relocate a large critical portion of the kernel without detection—a considerable, if not impossible effort. In situations where such an obstacle is absent, a general solution to this problem would involve monitoring individual shadow page table entries to ensure that they always point to specific, known good locations.

In addition, a notification should be sent by the trampoline only if an event happens, that is, an intruder should not be able to send bogus notifications. This could be done, for instance, by explicitly invoking the trampoline or jumping into arbitrary locations inside it (for instance, the VMCALL). Although this condition is not addressed by our prototype, we recognize it as a significant issue. One possible solution would be to first control the origin of branches by marking the memory region where the trampoline code resides with the non-execute (NX) bit. By doing so, every access to the trampoline would generate a fault, allowing us to check if the EIP value (the location from which the call was made) corresponds to an authorized hook location. If not, then we would know that the attacker is trying to make a bogus call to the trampoline and block it. The `cr2` register could also be monitored, which would allow us to check the destination of the branch, and enforce a single entry point into the trampoline code. This would in turn

block any attempts of branching into arbitrary locations of the trampoline code.

Other types of attacks can be avoided by disabling interrupts system-wide during the execution of the trampoline. This ensures that no other kernel thread is executed before the `VMCALL`. This guarantee, combined with our assumption that this system is running on a single processor, ensures that no one can change the monitored thread context to bypass the hook. Therefore, the code execution from the occurrence of the event up to the notification sent by the trampoline cannot be preempted.

This technique also automatically prevents A2 attacks, as interrupt disabling prevents an attacker from modifying any context information from the moment the event happens to the moment a response is received by the trampoline. Attack A5 is also prevented since the code responsible for carrying it out is already protected in the trampoline, and the fact that interrupts are disabled guarantees that its execution cannot be preempted. Although the triggering of non-maskable interrupts (NMIs) could circumvent the interrupt disabling and break the desired execution atomicity, we expect these to occur only during hardware fatal errors. By assuming the use of a single CPU core per VM in our prototype, we avoid the scenario in which an attacker could use a second core to explicitly send an NMI to the one running trampoline code and interrupt the execution flow. This assumption also prevents the occurrence of other race conditions involving multi-core architectures.

We acknowledge that some of the anti-circumvention techniques mentioned above are very specific to the type of hooking implemented. In particular, the kernel code and data structures that link the actual system call event to the SSDT hook itself, are relatively few and easy to protect, enabling us to create a protected chain. But in a more general scenario, where hooks can be placed in code or arbitrary data structures, creating an equivalent protection chain can be more complicated. By patching kernel code whose execution precedes the execution of a code hook, for instance, an attacker could jump around it. Existing solutions, such as SecVisor [32], could be integrated with our architecture to guarantee the kernel's code integrity and avoid this type of circumvention. Data hooks in arbitrary kernel data structures present a more interesting challenge because of data's volatile nature. But existing approaches like passive monitoring of kernel control data structures [25], mediation of changes to kernel data structures [38] and semantic integrity checking [24] could be used to raise the bar for an attacker. Although these techniques certainly help mitigate the more generic circumvention problem, their kernel-pervasive nature would add a significant performance impact to the overall architecture. In this case, a compromise between security and performance exists, which permits each application to make an appropriate tradeoff given its needs.

7 Related Work

Partitioning and Isolation The isolation properties provided by virtualization and their applications to security were first studied and formalized by Madnick and Donovan [19], Rushby [28], and Kelem [16]. More recently, Garfinkel et al. [8] have shown how such properties can be used to make OSes with different security requirements coexist in a single environment, by using a special-purpose hypervisor. Ta-Min et al. [36] went further by partitioning the OS system calls into an untrusted and a trusted domain and routing them according to a policy. Our work is based on a similar concept, except that we partition a single application between domains and protect the components placed in the untrusted one. Attention has also been given to isolation enforcement by controlling inter-VM communication. Sailer et al. [31] have integrated Mandatory Access Control inside the Xen hypervisor for such purpose. The ongoing XSM project [5] aims to create an access control framework inside Xen which can be used to deploy similar solutions.

Passive Monitoring The exposure of traditional security monitors to attackers has motivated the creation of isolated environments from where such monitoring can be done safely. Petroni et al. first proposed CoPilot, a coprocessor based kernel integrity monitor that runs in a PCI card [23] and monitors kernel memory using DMA. This work was later enhanced to include consistency checks at higher semantic levels [24]. Others use virtualization to accomplish similar goals without the need for special hardware. Garfinkel et al. proposed the technique of accessing and monitoring a system's memory state, which became known as virtual machine introspection [9]. This technique was later incorporated and further developed by other projects related to attack replaying [15], passive control-flow integrity checking [25], and intrusion detection [18]. Payne et al. establish security requirements for secure VM monitoring and describe the detailed mechanics of introspection [22]. Despite its clear usefulness and widespread adoption, introspection-based techniques are limited to passive system checks, which makes event-based monitoring extremely difficult. Our architecture uses memory introspection to complement our protected hooking architecture.

Memory Protection Introspection's passive nature implies that it can only detect, not actually prevent integrity violations. Although the former certainly has value, in certain situations the increased functionality provided by the later may be desirable. Microsoft's PatchGuard [20] attempts to protect certain key kernel data structures from modifications but fails to do so effectively. Their protection mechanism is based in the same domain as the attacker, and therefore prone to tampering. Seshadri et al. proposed a thin hypervisor which leverages the virtualization of memory to effectively protect kernel code and certain control flow transitions [32]. Xu et al. use a similar technique to implement

an intra-kernel access control framework with which access to kernel data structures can be controlled [38]. In Lares we also use hypervisor-based memory protection, but with different goals. Whereas the projects above aim to protect the entire kernel against certain classes of attacks, our protection infrastructure was built to protect the specific guest OS kernel components necessary for active monitoring. Moreover, we manage to provide fine-grained memory protection for the guest OS without the need for special hardware, unlike other approaches such as Mondrix [37].

Secure Code Execution Whereas memory protection focuses on the prevention of integrity violations to memory state, code attestation ensures the correct execution of code. Pioneer [33] implements verifiable code execution by carefully constructing a self-checking piece of code which is executed atomically inside an untrusted kernel. Our approach performs a similar task with the trampoline code by disabling interrupts system-wide while the hook notification is processed and sent. Our task is easier than Pioneer's, however, since the memory protections that we use prevent an intruder from modifying the code prior to its execution. Hardware support for trusted code execution has been recently introduced by the Intel Trusted Execution Technology (TxT) [11] and AMD Secure Virtual Machine (SVM) extensions [2], which provide stronger assurance for secure code execution as well as reduce the complexity of systems.

Active monitoring Active monitoring has always played an important role in systems security, due to its event-driven nature and potential for real-time attack prevention. Systems like anti-virus tools [35] and host-based intrusion detection systems [7] commonly hook the OS code/data and user applications to monitor events for suspicious behavior. Techniques such as system call interposition [26] and control flow integrity [1] can be used by such tools. Nevertheless, their fundamental limitation is exposure to attackers, which makes them prone to tampering. More recently, active monitors started leveraging the isolation provided by virtualization. These include systems like VMScope [12], which provides system call tracing ability. Dunlap et al. actively logs all I/O activity of a VM to enable VM backtracking and replay functionality, which can be used to analyze attacks [6]. Payne et al. intercept low-level guest disk activity at domain 0 to detect suspicious modifications to the file system [22]. Malware analysis tools [21, 39, 40] commonly make use of instruction-level active monitoring by running VMs on full-emulators like QEMU [4]. Taint-based malware analysis systems, such as Panorama [40], require the monitoring of individual instructions to be able to trace memory reads and writes. These techniques, nevertheless, are limited to low-level event monitoring, making them unsuitable for tools that require access to a higher abstraction level. Xenprobes [27] addresses this issue by providing a framework through which hooks can be placed in arbitrary locations inside a VM. But as their focus is systems management, they do not worry about protecting their hooks.

This analysis shows that none of the existing active monitoring solutions provide both a protected and flexible hooking infrastructure with acceptable performance. As event-driven security tools migrate into virtualized environments [34], the need for such solutions will grow.

8 Conclusions

Active monitoring is needed to support state-of-the-art host-based security applications such as intrusion detection and anti-virus tools. However, as recent research has focused on moving security applications into an isolated VM, the resulting architectures do not support active monitoring. Lares addresses this problem by giving security tools the ability to do active monitoring while still benefiting from the increased security of an isolated virtual machine. Our security analysis shows that Lares provides security suitable for deployment on production systems. And our performance evaluation shows that Lares' overall impact on system performance is small. The Lares architecture is generally applicable to any application that requires secure active monitoring.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0627430. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Abadi, M. Budiu, and Ú. E. J. Ligatti. Control-Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

[2] Advanced Micro Devices. Amd64 architecture programmer's manual volume 2: System programming, 2007.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Computer Security and Privacy*, 1997.

[4] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[5] G. Coker. Xen security modules (XSM). Presented at the 2007 Xen Summit.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2002.

[7] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of the*

IEEE Symposium on Research in Security and Privacy, Los Alamitos, CA, USA, 1994.

[8] 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.

[9] 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.

[10] G. Hoglund. *Rootkits: Subverting the Windows Kernel*. Addison Wesley, 2005.

[11] Intel Corporation. Intel trusted execution technology, 2007.

[12] X. Jiang and X. Wang. "Out-of-the-Box" monitoring of VM-based high-interaction honeypots. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2007.

[13] X. Jiang, D. Xu, and X. Wang. Stealthy malware detection through VMM-based "Out-of-the-Box" semantic view reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[14] 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*, June 2006.

[15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[16] 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.

[17] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, 2006.

[18] 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.

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

[20] Microsoft Corporation. Kernel patch protection: Frequently asked questions. `http://www.microsoft.com/whdc/driver/kernel/64bitpatch_FAQ.mspx`.

[21] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium of Security and Privacy*, 2007.

[22] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the Annual Computer Security Applications Conference*, 2007.

[23] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[24] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the USENIX Security Symposium*, 2006.

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

[26] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[27] N. A. Quynh and K. Suzaki. Xenprobe: A lightweight user-space probing framework for xen virtual machine. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

[28] J. M. Rushby. Proof of separability: A verification technique for a class of security kernels. *Lecture Notes in Computer Science*, 137:352 – 357, April 1982.

[29] J. Rutkowska. Rootkit hunting vs. compromise detection. In *Proceedings of Black Hat Federal*, 2006.

[30] J. Rutkowska. Subverting vista kernel for fun and profit. In *Proceedings of Black Hat USA*, 2006.

[31] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a mac-based security architecture for the xen opensource hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, December 2005.

[32] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the ACM Symposium on Operating System Principles*, 2007.

[33] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2005.

[34] Symantec Corporation. Symantec Virtual Security Solution and PCs with Intel vPro Technology, 2007.

[35] P. Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[36] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.

[37] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: Memory Isolation for Linux using Mondriaan Memory Protection. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2005.

[38] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a vmm-based usage control framework for os kernel integrity protection. In *Proceedings of the Symposium on Access control Models and Technologies*, 2007.

[39] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, February 2008.

[40] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference of Computer and Communication Security*, 2007.